

**IN THE UNITED STATES PATENT AND TRADEMARK OFFICE  
BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES**

Applicant: Rebecca A. Kocot

Serial No.: 10/664,636

Filed: September 19, 2003

Title: USER INTERFACE SOFTWARE DEVELOPMENT TOOL AND  
METHOD FOR ENHANCING THE SEQUENCING OF INSTRUCTIONS  
WITHIN A SUPERSCALAR MICROPROCESSOR PIPELINE BY  
DISPLAYING AND MANIPULATING INSTRUCTIONS IN THE  
PIPELINE

Grp./A.U.: 2193

Examiner: Insun Kang

Confirmation No.: 5055

Commissioner for Patents  
P.O. Box 1450  
Alexandria, VA 22313-1450

Mail Stop Appeal Brief-Patents

I hereby certify that this correspondence is being electronically filed with United States Patent and trademark Office on:	
<u>August 24, 2009</u>	(Date)
<u>Debbie Sams</u>	
(Printed or typed name of person signing the certificate)	
<u>/Debbie Sams/</u>	
(Signature of the person signing the certificate)	

ATTENTION: Board of Patent Appeals and Interferences

Sirs:

**APPEAL BRIEF UNDER 37 C.F.R. §41.37**

This is an appeal from a Final Rejection dated January 22, 2009, of Claims 1 and 3-20. The Appellants submit this Brief with the statutory fee of \$540.00 as set forth in 37 C.F.R. §41.20(b)(2), and hereby authorize the Commissioner to charge any additional fees connected with this communication or credit any overpayment to Deposit Account No. 08-2395.

This Brief contains these items under the following headings, and in the order set forth below in accordance with 37 C.F.R. §41.37(c)(1):

- i. REAL PARTY IN INTEREST
- ii. RELATED APPEALS AND INTERFERENCES
- iii. STATUS OF CLAIMS
- iv. STATUS OF AMENDMENTS
- v. SUMMARY OF CLAIMED SUBJECT MATTER
- vi. GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL
- vii. APPELLANTS' ARGUMENTS
- viii. APPENDIX A - CLAIMS
- ix. APPENDIX B - EVIDENCE
- x. RELATED PROCEEDINGS APPENDIX

**i) REAL PARTY IN INTEREST**

The real party in interest in this appeal is the Assignee, LSI Logic Corporation.

**ii) RELATED APPEALS AND INTERFERENCES**

Appellant does not know of any prior and pending Appeals, Interferences, or Judicial Proceedings directly related to, affecting, affected by, or having a bearing on the Board's decision in this appeal.

**iii) STATUS OF THE CLAIMS**

Claims 1 and 3-20 are rejected, Claim 2 is canceled.

Herein, all rejections of Claims 1 and 3-20 are being appealed.

**iv) STATUS OF THE AMENDMENTS**

No amendments have been made in response to the Office Action of January 22, 2009 (hereinafter "Office Action") or the Advisory Action of April 6, 2009 (hereinafter "Advisory Action") and no amendments are pending.

**v) SUMMARY OF CLAIMED SUBJECT MATTER**

Independent Claim 1 features a graphics rendering engine comprising a sequence of instruction addresses for display on a screen accessible to a user. The screen comprises a graphical user interface (GUI) for receiving user input to select one of the instruction addresses. The graphics rendering engine also comprises a sequence of processor pipeline stages attributable to respective

ones of the sequence of instruction addresses. When the user input is received by the GUI, the screen displays a designator for at least one of the instruction addresses to denote that a corresponding designated instruction address will proceed to a succeeding stage in the processor pipeline during a next clock cycle and a non-designator for another one of at least one of the instruction addresses to denote that a corresponding non-designated instruction address will not proceed to a succeeding stage in the processor pipeline during the next clock cycle. (*See, e.g.*, lines 18-26, page 4 of the original specification.)

Independent Claim 8 is directed to a software development tool that uses the graphics rendering engine described in independent Claim 1 to receive a first sequence of addresses where a user selects a particular instruction address within the first sequence of instruction addresses shown in a particular stage of a processor pipeline with the same designator and non-designator as described in independent Claim 1. The software development tool of independent Claim 8 also includes an instruction address field which can be used to select a particular instruction address within the first sequence and move that instruction address to another within the first sequence. Also included in the software development tool of Claim 8 is a scheduler that responds to the moved instruction address to form a second sequence of instruction addresses that has a higher instruction throughput in the processor pipeline than the first sequence of instruction addresses. (*See, e.g.*, line 28 on page 4 through line 3 on page 5 of the original specification.)

Independent Claim 16 features a method for displaying a progression of instruction addresses through a processor pipeline comprising selecting a breakpoint by a user at a breakpoint location on a display screen to select an instruction address within the same line as the breakpoint and a clock cycle associated with the selected instruction address. All instruction addresses within the processor

pipeline that will proceed to a succeeding stage in the processor pipeline are designated and all instruction addresses within the processor pipeline that will not proceed to the succeeding stage are not designated. (*See, e.g.*, lines 5-16 on page 5 of the original specification.)

**vi) GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL**

(A) Whether Claims 1 and 4-7 are obvious over the combination of U.S. Patent Application Publication No. 2003/0110476 by Aihara (hereinafter “Aihara”) in view of a paper entitled “Visualizing Application Behavior on Superscalar Processors” by Stolte, *et al.* (hereinafter “Stolte”) as applied by the Office Action at pages 2-4.

(B) Whether Claims 8-10 and 12-15 are obvious over the combination of Aihara, U.S. Patent No. 5,913,052 to Beatty, *et al.* (hereinafter “Beatty”), and Stolte as applied by the Office Action at pages 4-7.

(C) Whether Claims 16-20 are obvious over Aihara in view of Beatty as applied by the Office Action at pages 7-9.

(D) Whether Claim 3 is obvious over the combination of Aihara, Stolte, and U.S Patent Application Publication No. 2002/0130871 by Hill, *et al.* (hereinafter “Hill”) as applied by the Office Action at page 9.

(E) Whether Claim 11 is obvious over the combination of Aihara, Beatty, Stolte, and Hill as applied by the Office Action at page 10.

## **vii) APPELLANTS' ARGUMENT**

**(A) In Grounds of Rejection (A), the obviousness rejection of Claims 1 and 4-7 over Aihara and Stolte are improper.**

### **Claim 1**

**1) The obviousness rejection is improper because it relies on Stolte to teach features that not taught in the cited parts of Stolte.**

Claim 1 recites:

...a sequence of instruction addresses for display upon a screen accessible to a user, wherein the screen comprises a graphical user interface (GUI) for receiving user input to select one of the instruction addresses... (Emphasis added.)

In the Office Action, the Examiner states:

“...Aihara does not explicitly teach that the screen comprises a graphical user interface (GUI) for receiving user input to select one of the instruction addresses. However, Stolte teaches a pipeline visualization system that includes a GUI for user manipulation of the pipeline instructions (i.e. page 5, left col., first paragraph)...”

(*See* Office Action, page 2.)

Stolte teaches, in Fig. 2, a pipeline view is generated that shows all instructions in a pipeline.

As the Examiner points out, Stolte teaches that a user controls a pipeline animation using controls similar to those on a VCR enabling a user to single-step through the pipeline. However, Stolte does not teach or suggest enabling a user to select a single instruction address out of all the instruction addresses in the view. On the contrary, the user must animate and step through each instruction addresses, one instruction address at a time, to arrive at a selected instruction address. This could take several steps to arrive at the single instruction address that the user wants to view. The invention as presently claimed teaches that the user can go directly to the instruction address desired in one step by highlighting the instruction address.

As such, the cited portions of Aihara and Stolte do not teach or suggest a sequence of instruction addresses for display upon a screen accessible to a user, wherein the screen comprises a

graphical user interface (GUI) for receiving user input to select one of the instruction addresses as recited in pending Claim 1. For that reason, the Office Action has not provided a *prima facie* case of obviousness for pending Claim 1.

#### Claims 4-7

Claims 4-7 are non-obvious over the combination of Aihara and Stolte, as applied by the Office Action, at least, by their dependence on independent Claim 1.

**(B) In Grounds of Rejection (B), the obviousness rejection of Claims 8-10 and 12-15 over Aihara, Beatty, and Stolte are improper.**

#### Claim 8

**1) The obviousness rejection is improper because it relies on Stolte to teach features that are not taught in the cited part of Stolte.**

Claim 8 recites:

...an instruction address field that, upon selection by a user via the pointing device, allows the user to move said another at least one instruction address...

In the Office Action, the Examiner states:

..."Aihara does not explicitly teach that an instruction address field that, upon selection by a user via the pointing device, allows the user to move said at least one instruction address. However, Stolte teaches a pipeline visualization system that includes a GUI for user manipulation of the pipeline instructions (i.e. page 5, left col., first paragraph)..."

(See Office Action, page 6.)

As established above, the cited portion of Stolte does not allow a user to select a single instruction address but, rather, a range of instruction addresses that the user must single step through. Thus, for the same reasons given above, the cited portions of the cited combination of Aihara, Beatty, and Stolte, as applied by the Examiner, do not teach or suggest an instruction address field that, upon selection by a user via the pointing device, allows the user to move said at least one other

instruction address as recited in pending independent Claim 8. As such, the combination of Aihara, Beatty, and Stolte does not provide a *prima facie* case of obviousness for pending Claim 8.

**2) The obviousness rejection is improper because it relies on both Aihara and Stolte to teach features that are not taught in either the cited parts of Aihara or Stolte.**

Claim 8 recites:

...a scheduler that responds to the moved said another at least one instruction address to form a second sequence of instruction addresses that has a higher instruction throughput in the processor pipeline than the first sequence of instruction addresses.

In the Office Action, the Examiner states:

“Aihara and Stolte further discloses: a scheduler that responds to the moved said another at least one instruction address to form a second sequence of instructions that has a higher instruction throughput in the processor pipeline than the first sequence of instructions (i.e. Aihara, page 5, 0063; Stolte, page 6, right col., second paragraph).

The cited portion of Aihara teaches, in Fig. 14a, an indication of which stages are stalled. However, the Appellant fails to find where an indication of a stalled stage teaches or suggests a scheduler that responds to a moved instruction address to form a second sequence of instruction addresses. Furthermore, since the cited portion of Aihara does not teach or suggest a second sequence of instruction addresses, the cited portion can not teach or suggest second sequence of instruction addresses has a higher throughput than a first sequence of instruction addresses.

The cited portion of Stolte states:

“We can now use the source code view to correlate this pipeline behavior with the application’s source code: the application is executing a tight loop of floating point arithmetic. With this information, the programmer can now attempt to restructure the code to reduce the number of dependencies or interleave other code into the loop to better utilize the processor.”



Again, as with the cited portion of Aihara, the Appellant fails to find any teaching or suggestion of a second sequence of instruction addresses or that a second sequence of instruction addresses has a higher instruction throughput than a first sequence of instruction addresses.

As such, neither the cited portion of Aihara nor the cited portion of Stolte, as relied upon by the Examiner, teaches or suggests a scheduler that responds to the moved said another at least one instruction address to form a second sequence of instruction addresses that has a higher instruction throughput in the processor pipeline than the first sequence of instruction addresses as recited in independent Claim 8. Therefore, the cited combination of Aihara, Beatty, and Stolte does not provide a *prima facie* case of obviousness for pending independent Claim 8.

#### Claims 9-10 and 12-15

Claims 9-10 and 12-15 are non-obvious over the combination of Aihara, Beatty, and Stolte, as applied by the Examiner, at least, by their dependence on independent Claim 8.

**(C) In Grounds of Rejection (C), the obviousness rejection of Claim 16-20 over Aihara and Beatty are improper.**

#### Claim 16

**1) The obviousness rejection is improper because it relies on Aihara to teach features that are not taught in the cited part of Aihara.**

Claim 16 recites:

...selecting a breakpoint within a breakpoint column of a display screen, via user input at a breakpoint location on the display screen, to select:...a clock cycle associated with the selected instruction address being in a stage with in the processor pipeline...

In the Office Action, the Examiner states:

“Aihara in view of Beatty further discloses:

- a clock cycle associated with the selected instruction address being in a stage within the processor pipeline (i.e. page 3, 0046)...”

(See Office Action, page 8.)

The cited portion of Aihara states

“Here, the “cycle-accurate instruction set simulator” refers to an instruction set simulator which simulates an operation of a processor not by the instruction, but by the stage such as fetch (F)/decode (D)/execute (E)/memory access (M)/write back (W) or the like, while taking the pipeline processing into consideration. In the following, “address information of the instruction in execution at each stage of the pipeline” refers to the addresses being executed by the respective stages of (F)/decode (D)/execute (E)/memory access (M)/write back (W) and the like. Moreover, “address information of the program in execution” refers to an address of a source code initiated for execution. In the cycle-accurate instruction set simulator, the address information of the program in execution refers to the address in which the fetch (F) stage is initiated.”

Thus, the cited portion of Aihara teaches a simulator that simulates an operation by stage.

However, the Appellant fails to find where the cited portion of Aihara relied upon by the Examiner teaches or suggests selecting a breakpoint to select a clock cycle associated with a selected instruction address being in a stage within a processor pipeline. It appears the cited portion of Aihara is concerned with the stage (fetch, decode, execute, *etc.*) of an instruction address, but not the clock cycle of a selected instruction address.

As such, the cited combination of Aihara and Beatty do not teach or suggest selecting a breakpoint within a breakpoint column of a display screen, via user input at a breakpoint location on the display screen, to select a clock cycle associated with the selected instruction address being in a stage within the processor pipeline as recited in pending Claim 16. Therefore, the cited portions of the cited combination of Aihara and Beatty, as applied by the Examiner, do not establish a *prima facie* case of obviousness for pending Claim 16.

### Claims 17-20

Claims 17-20 are non-obvious over the combination of Aihara and Beatty, as applied by the Examiner, at least, by their dependence on independent Claim 16

**(D) In Grounds of Rejection (D), the obviousness rejection of Claim 3 over Aihara, Stolte, and Hill is improper**

Claim 3 is non-obvious over the combination of references, as applied by the Office Action, at least, by its dependence on independent Claim 1.

**(E) In Grounds of Rejection (E), the obviousness rejection of Claim 11 over Aihara, Beatty, Stolte, and Hill is improper.**

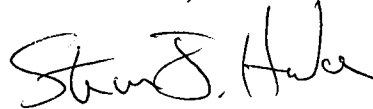
Claim 11 is non-obvious over the combination of references, as applied by the Office Action, at least, by its dependence on independent Claim 8.

## **CONCLUSIONS**

For the reasons set forth above, allowance of all the claims presently in the application is respectfully requested, as is passage to issuance of the present application.

Respectfully submitted,

**HITT GAINES, P.C.**

A handwritten signature in black ink, appearing to read "Steven J. Hanke", written in a cursive style.

Steven J. Hanke  
Registration No. 58,076

Dated: August 24, 2009

Hitt Gaines, P.C.  
P.O. Box 832570  
Richardson, Texas 75083-2570  
(972) 480-8800  
(972) 480-8865 (Fax)  
steve.hanke@hittgaines.com

## **vii) APPENDIX A - CLAIMS**

1. (Previously Presented) A graphics rendering engine stored on a computer-storage medium and executable by a computer, the graphics rendering engine comprising:

a sequence of instruction addresses for display upon a screen accessible to a user, wherein the screen comprises a graphical user interface (GUI) for receiving user input to select one of the instruction addresses;

a sequence of processor pipeline stages attributable to respective ones of the instruction addresses, wherein during times when the user input is received by the GUI, the screen displays:

a designator for at least one of the instruction addresses to denote that a corresponding designated instruction address will proceed to a succeeding stage in the processor pipeline during a next clock cycle; and

a non-designator for another one of at least one of the instruction addresses to denote that a corresponding non-designated instruction address will not proceed to a succeeding stage in the processor pipeline during the next clock cycle.

2. (Canceled)

3. (Original) The graphics rendering engine as recited in claim 1, wherein the screen comprises a pop-up window.

4. (Original) The graphics rendering engine as recited in claim 1, wherein the designator is a color that highlights the stage attributable to the at least one instruction that will proceed to the succeeding stage.

5. (Original) The graphics rendering engine as recited in claim 4, wherein the color differs depending on which stage is highlighted.

6. (Previously Presented) The graphics rendering as recited in claim 1, wherein the processor pipeline is a pipeline of a superscalar processor where more than one instruction can exist within each stage of the pipeline.

7. (Previously Presented) The graphics rendering engine as recited in claim 1, wherein the user actuates a pointing device to supply the user input to the GUI for selecting only one of the instruction addresses, wherein in response to said selection, the screen displays the designator over a field bearing a stage name for all of the instruction addresses that will proceed to the next stage in the processor pipeline.

8. (Previously Presented) A software development tool stored on a computer-storage medium and executable by a computer, the software development tool comprising:

source code represented as a first sequence of instruction addresses;

a graphics rendering engine coupled to receive the first sequence of instruction addresses and produce a graphical user interface (GUI) window that includes:

a breakpoint field that, upon receiving user input via a pointing device:

selects a particular instruction address within the first sequence of instruction addresses shown in a particular stage of a processor pipeline;

displays all instruction addresses within the first sequence of instruction addresses along with corresponding stages of a processor pipeline during a clock cycle in which the particular instruction address is within the particular stage;

assigns a designator to at least one instruction address of the first sequence of instruction addresses to denote that a corresponding designated instruction will proceed to a succeeding stage in the processor pipeline during a clock cycle succeeding the clock cycle; and

assigns a non-designator to another at least one instruction address of the first sequence of instruction addresses to denote that a corresponding non-designated instruction will not proceed to a succeeding stage in the processor pipeline during a clock cycle succeeding the clock cycle;

an instruction address field that, upon selection by a user via the pointing device, allows the user to move said another at least one instruction address; and

a scheduler that responds to the moved said another at least one instruction address to form a second sequence of instruction addresses that has a higher instruction throughput in the processor pipeline than the first sequence of instruction addresses.

9. (Previously Presented) The software development tool as recited in claim 8, wherein the graphics rendering engine further displays all instructions within the first sequence of instruction addresses and assigns a designator to a number of the instruction addresses of the second sequence of instruction addresses that exceeds a number of the at least one instruction addresses of the first sequence of instruction addresses.

10. (Previously Presented) The software development tool as recited in claim 8, wherein the second sequence of instruction addresses requires fewer clock cycles through the processor pipeline than the first sequence of instruction addresses.

11. (Original) The software development tool as recited in claim 8, wherein the window comprises a pop-up window rendered upon a computer display screen.

12. (Previously Presented) The software development tool as recited in claim 8, wherein the designator is a color that highlights the stage attributable to the at least one instruction address that will proceed to the succeeding stage.

13. (Original) The software development tool as recited in claim 12, wherein the color differs depending on which stage is highlighted.

14. (Previously Presented) The software development tool as recited in claim 8, wherein the processor pipeline is a pipeline of a superscalar processor where more than one instruction can exist within each stage of the pipeline.

15. (Previously Presented) The software development tool as recited in claim 8, wherein the user actuates the pointing device to select the particular instruction address and, in response thereto, the window displays the designator over a stage number field bearing a stage name for all of the first sequence of instruction addresses that will proceed to the succeeding stage in the processor pipeline.

16. (Previously Presented) A method for displaying progression of instruction addresses through a processor pipeline, comprising:

selecting a breakpoint within a breakpoint column of a display screen, via user input at a breakpoint location on the display screen, to select:

an instruction address within the same line as the breakpoint; and



a clock cycle associated with the selected instruction address being in a stage within the processor pipeline;

designating all instruction addresses within the processor pipeline that will proceed to a succeeding stage of the processor pipeline; and

not designating all instruction addresses within the processor pipeline that will not proceed to a succeeding stage of the processor pipeline.

17. (Previously Presented) The method as recited in claim 16, wherein said designating comprises receiving a signal from a stage debug register by a graphics rendering engine to denote that the instruction addresses being designated will proceed to the succeeding stage of the processor pipeline.

18. (Original) The method as recited in claim 16, wherein said designating comprises checking resources of a processor to determine if the instruction addresses will be allowed to proceed and, if so, sending a signal from a debug register that stores the checking outcome to designate the instruction addresses that have corresponding resources available to allow such instruction addresses to proceed.

19. (Previously Presented) The method as recited in claim 16, wherein said designating comprises highlighting the designated instruction addresses with a color different from the background color of the display screen.

20. (Previously Presented) The method as recited in claim 16, wherein said designating comprises highlighting the stage corresponding to the designated instruction addresses with a color, and wherein the color differs depending on which stage is highlighted.

## **ix) APPENDIX B - EVIDENCE**

The evidence in this appendix includes a U.S. Patent to Beatty, U.S. Patent Application Publications by Aihara and Hill, and a paper by Stolte. The U.S. Patent Application Publication by Hill was entered in the record by the Examiner with the Office Action of June 27, 2007. The U.S. Patent to Beatty and the U.S. Patent Application Publication by Aihara was entered in the record by the Examiner with the Office Action of December 28, 2007. The paper by Stolte was entered in the record by the Examiner with the Office Action of July 24, 2008.

**x) RELATED PROCEEDINGS APPENDIX**

NONE



US005913052A

**United States Patent** [19][11] **Patent Number:** **5,913,052****Beatty et al.**[45] **Date of Patent:** **Jun. 15, 1999**

[54] **SYSTEM AND METHOD FOR DEBUGGING  
DIGITAL SIGNAL PROCESSOR SOFTWARE  
WITH AN ARCHITECTURAL VIEW AND  
GENERAL PURPOSE COMPUTER  
EMPLOYING THE SAME**

5,680,584 10/1997 Herdeg et al. .... 395/500

*Primary Examiner*—Kevin J. Teska  
*Assistant Examiner*—Ayni Mohamed

[57] **ABSTRACT**

A system and method, operable on a general purpose computer, for debugging software that is to control a digital signal processor ("DSP") and a general purpose computer employing either the system or the method. The present invention is employable with either a real DSP or an emulated DSP. When the DSP is emulated, the system includes: (1) architectural display circuitry that displays an architecture of a particular DSP in a window on a display of the general purpose computer, the architecture including a graphical device layout and at least one field corresponding to a register of the DSP and (2) software simulation circuitry that employs a processor of the general purpose computer to simulate operation of DSP software and emulate operation of the particular DSP to cause the particular DSP to change states over time, the architectural display circuitry updating the at least one field to reflect changes in the states, the architectural display circuitry and the software simulation circuitry cooperating to allow a user to debug the software by visually inspecting the graphical device layout and the at least one field.

[75] Inventors: **Paul E. Beatty**, Leesport; **Paul G. D'Arcy**, North Wales; **Lee E. Deschler**; **Mohit K. Prasad**, both of Bethlehem, all of Pa.

[73] Assignee: **Lucent Technologies Inc.**, Murray Hill, N.J.

[21] Appl. No.: **08/788,754**

[22] Filed: **Jan. 24, 1997**

[51] **Int. Cl.<sup>6</sup>** ..... **G06F 9/455**

[52] **U.S. Cl.** ..... **395/500**

[58] **Field of Search** ..... 364/578; 395/500

[56] **References Cited**

**U.S. PATENT DOCUMENTS**

5,220,512 6/1993 Watkins et al. .... 364/488  
5,325,309 6/1994 Halaviati et al. .... 364/488  
5,623,418 4/1997 Rostoker et al. .... 364/488

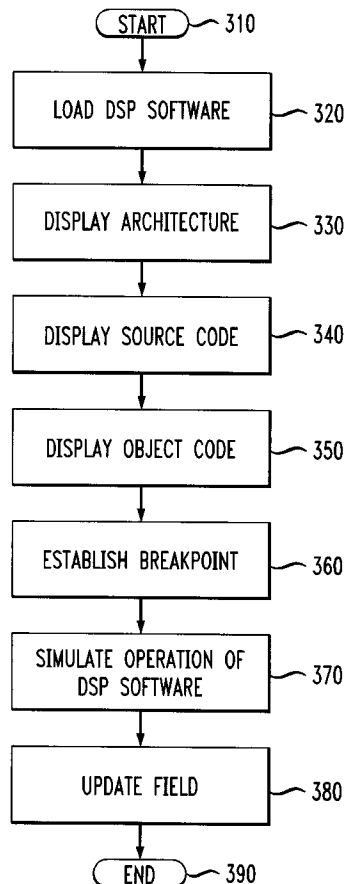
**40 Claims, 10 Drawing Sheets**

FIG. 1

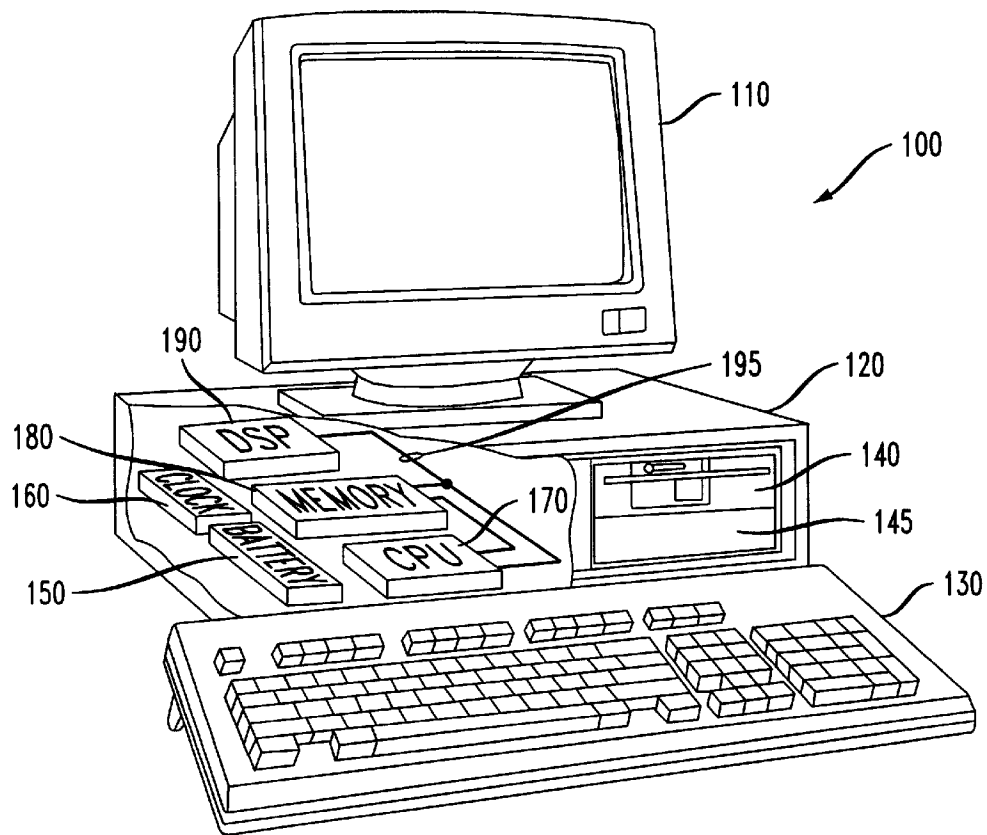
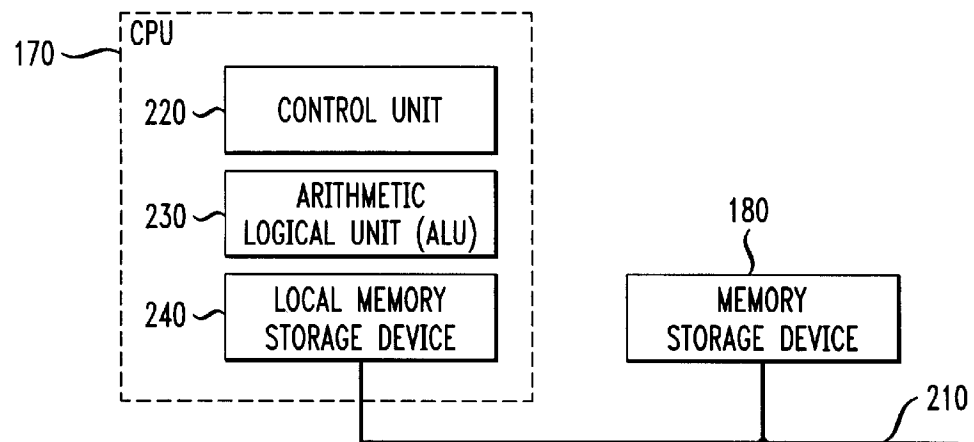
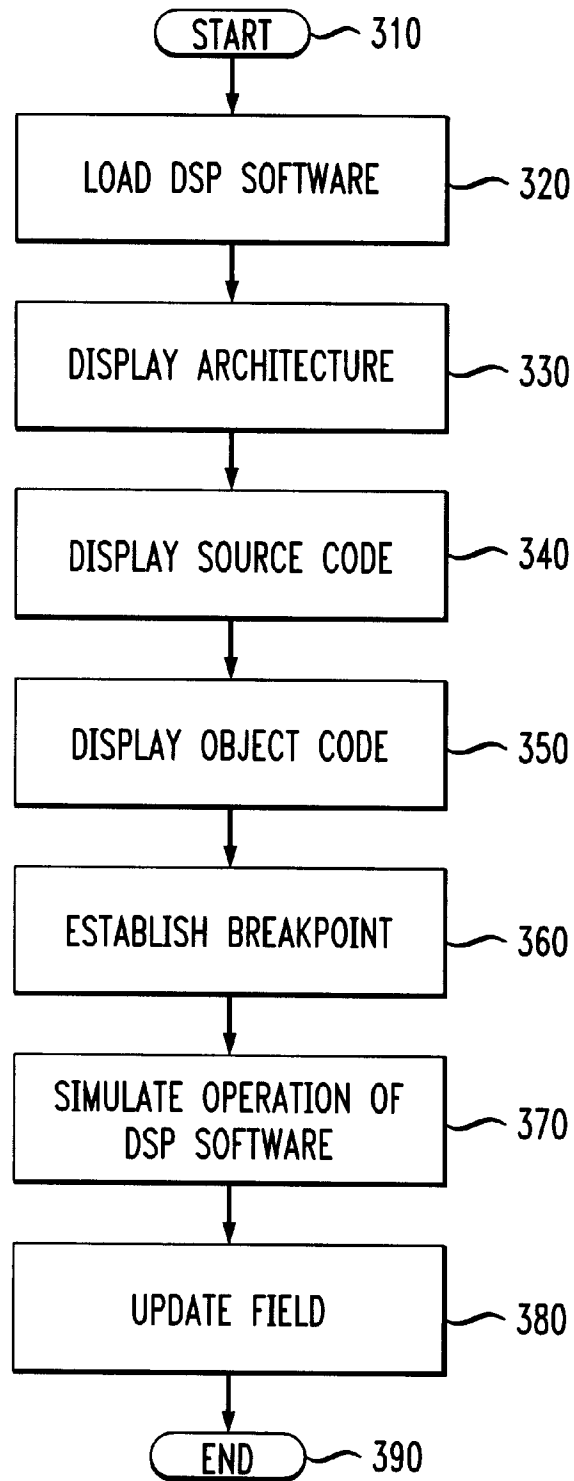


FIG. 2



*FIG. 3*

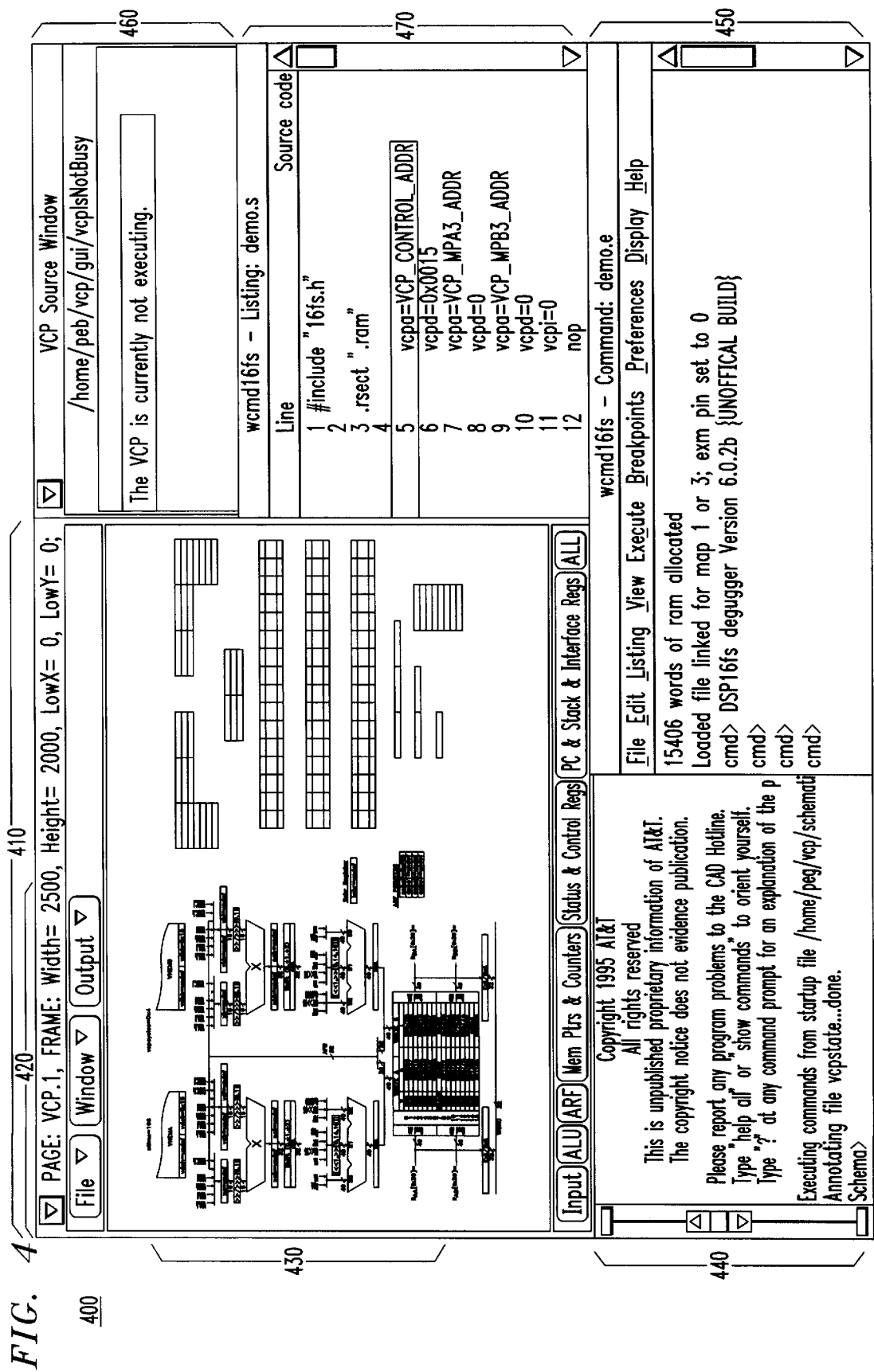


FIG. 5

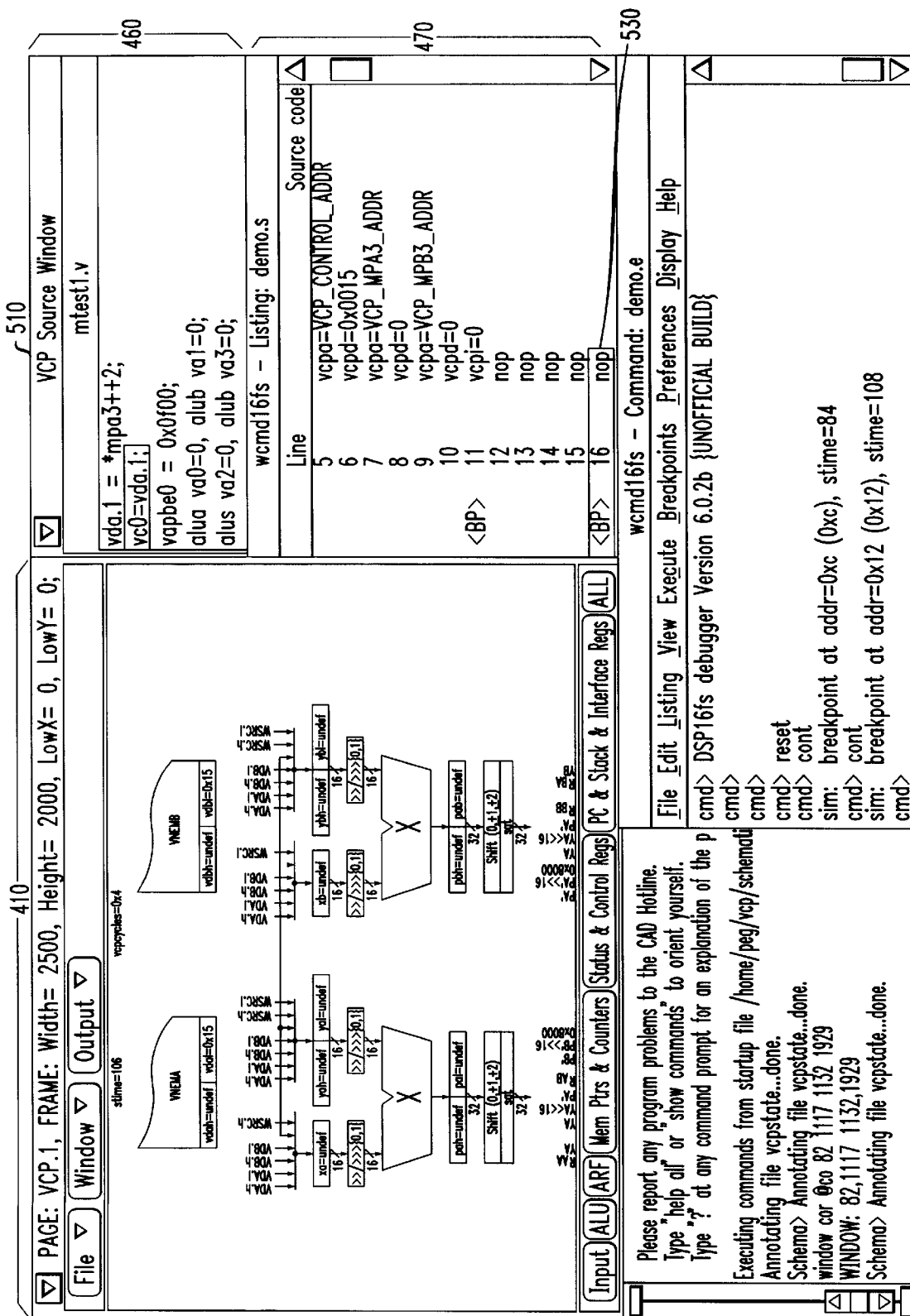
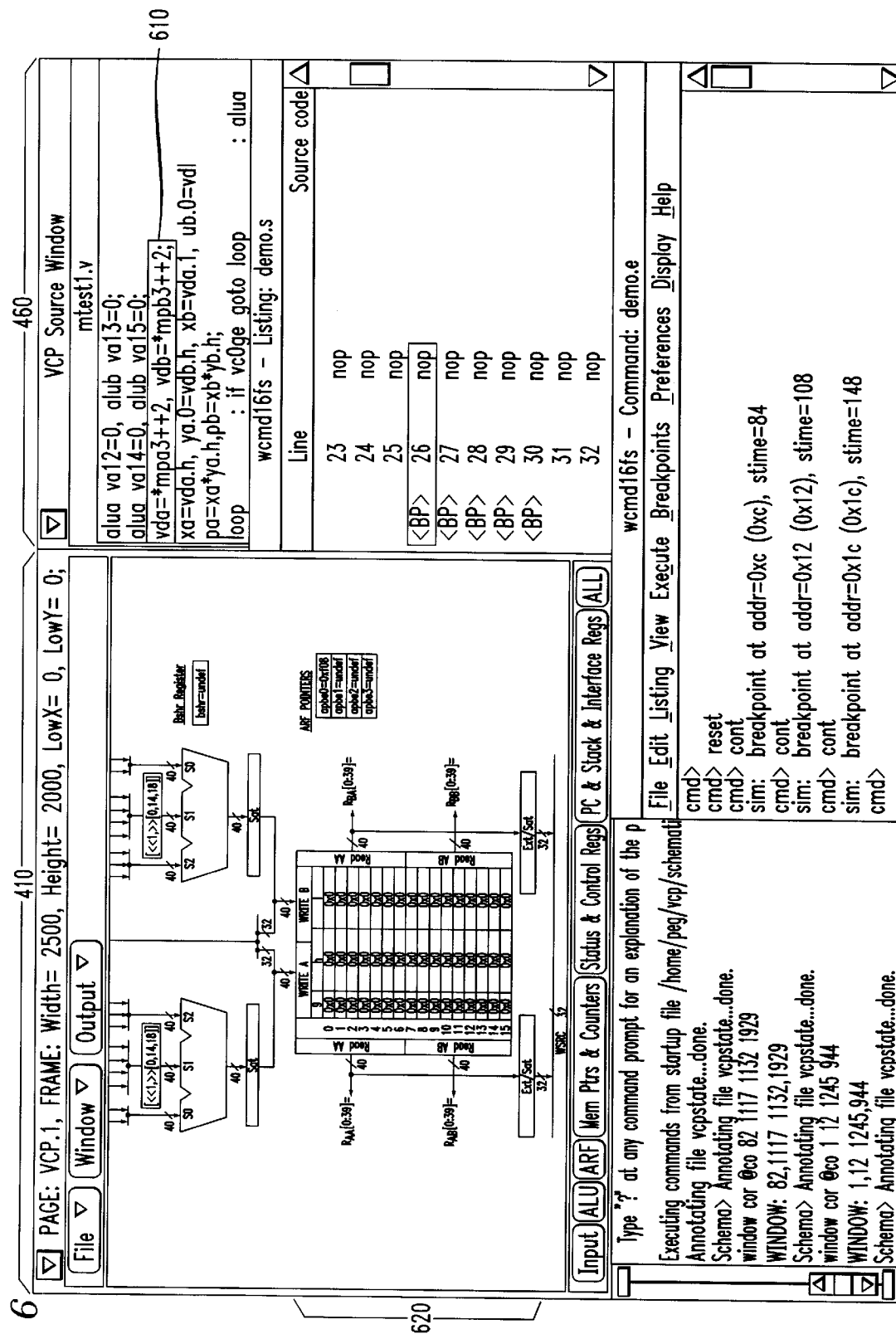
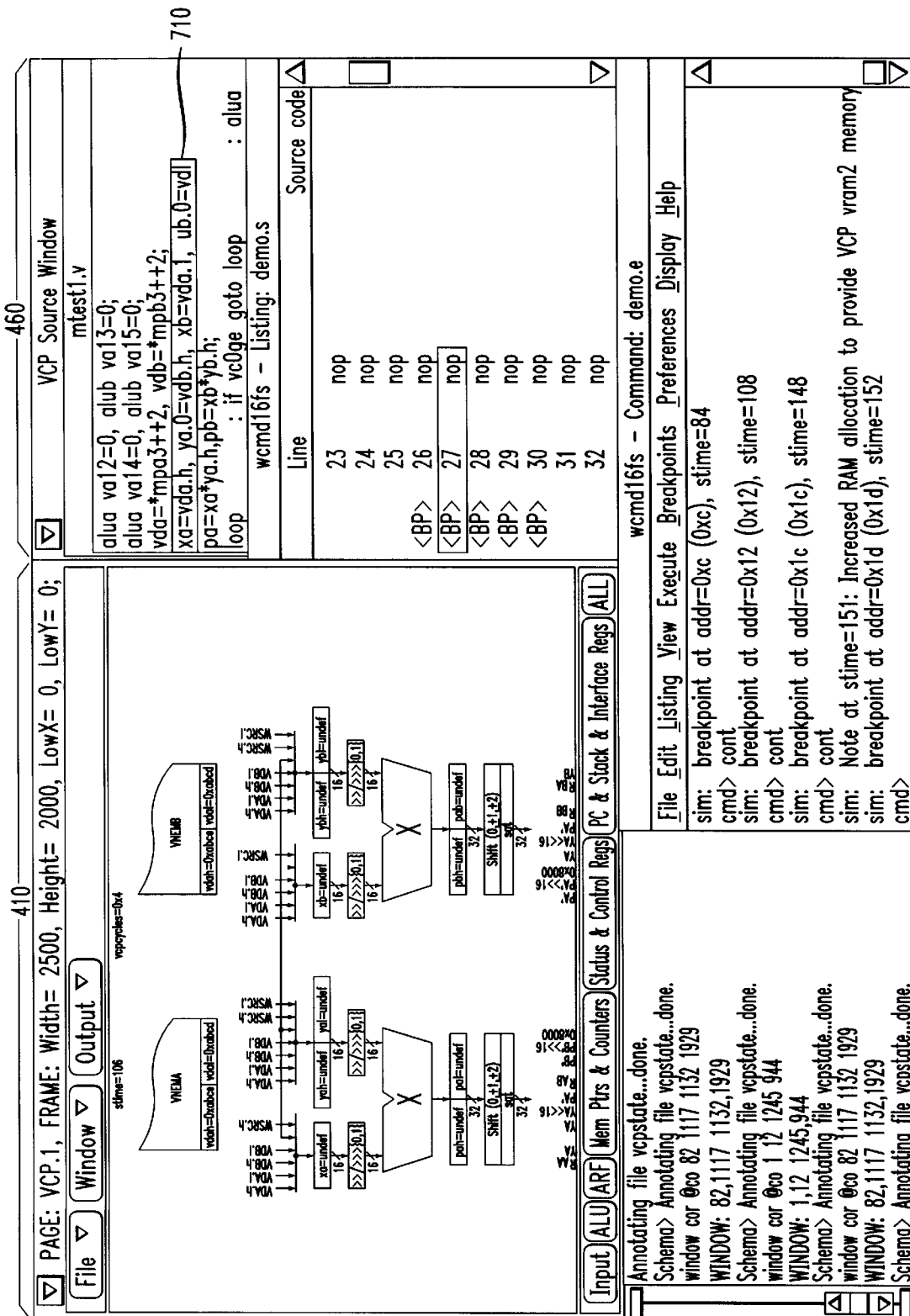


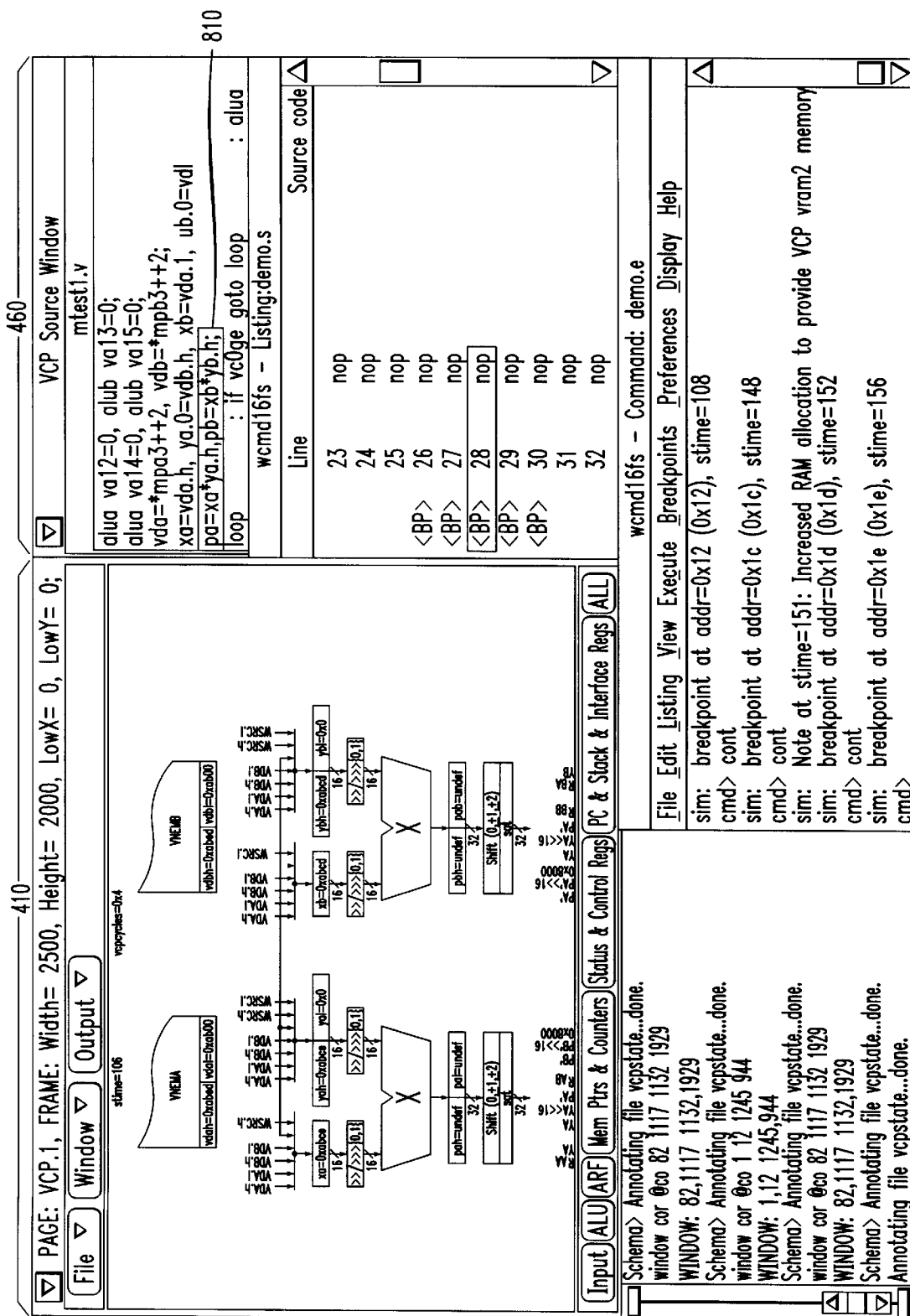


FIG. 6

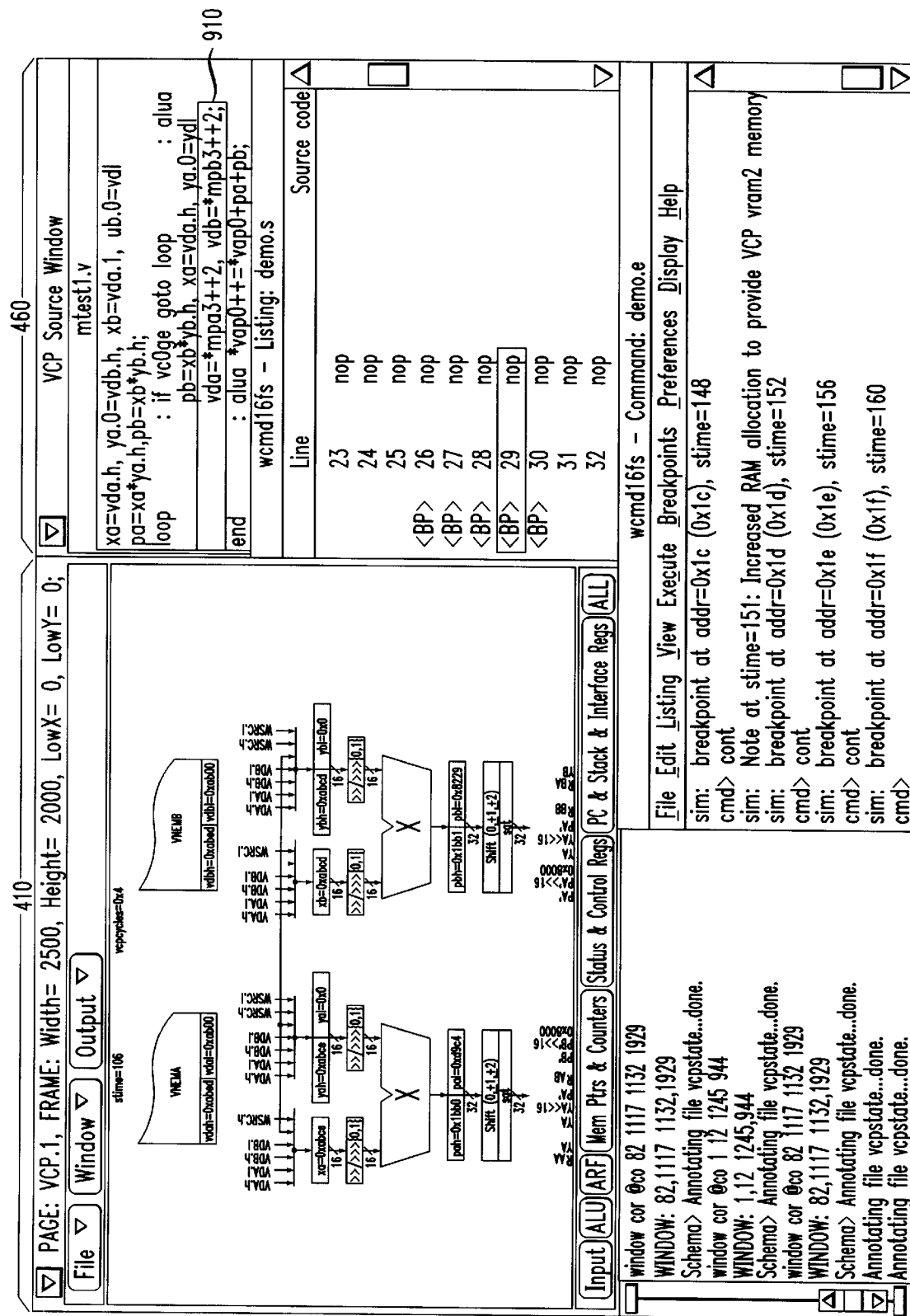


2





9



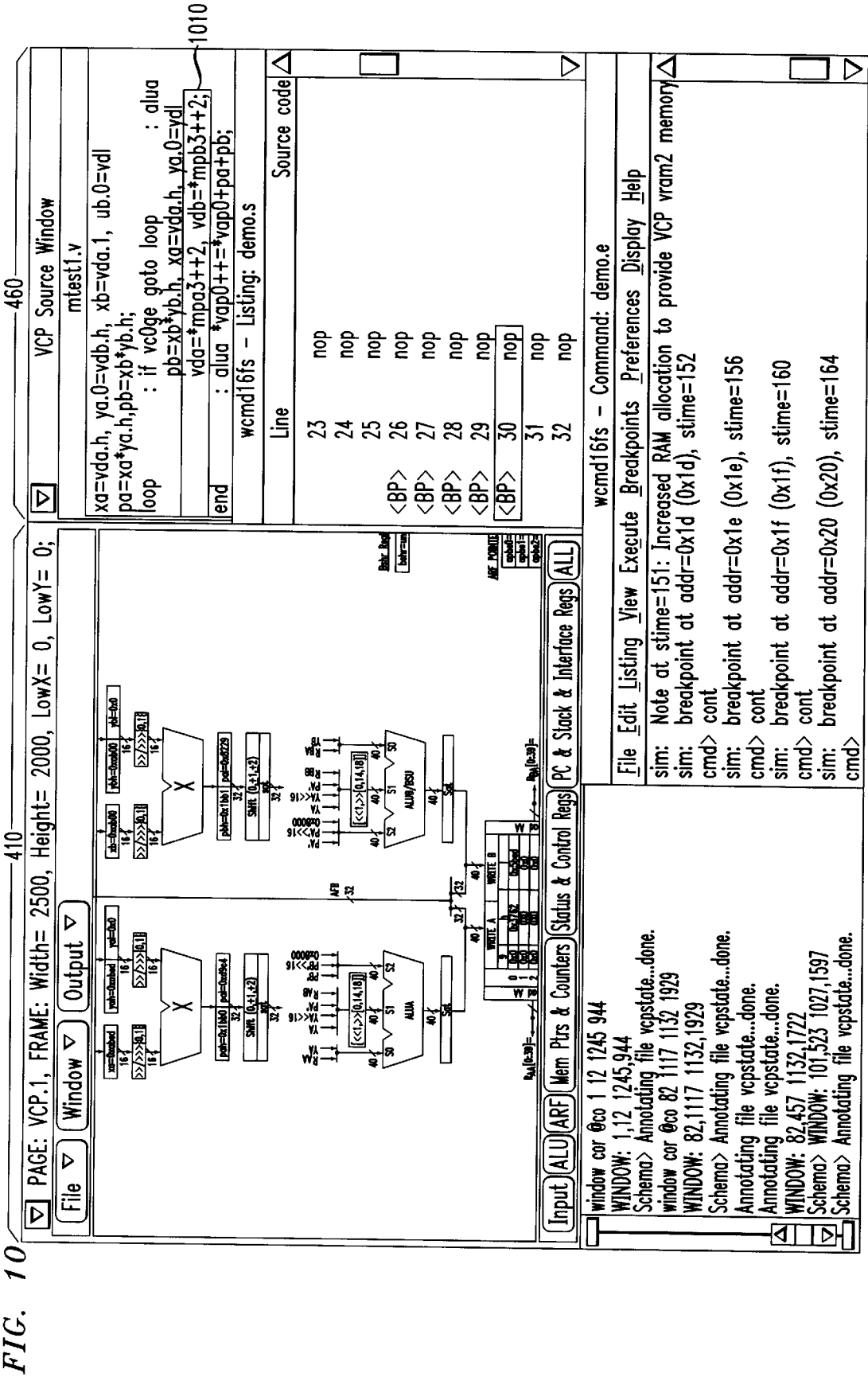
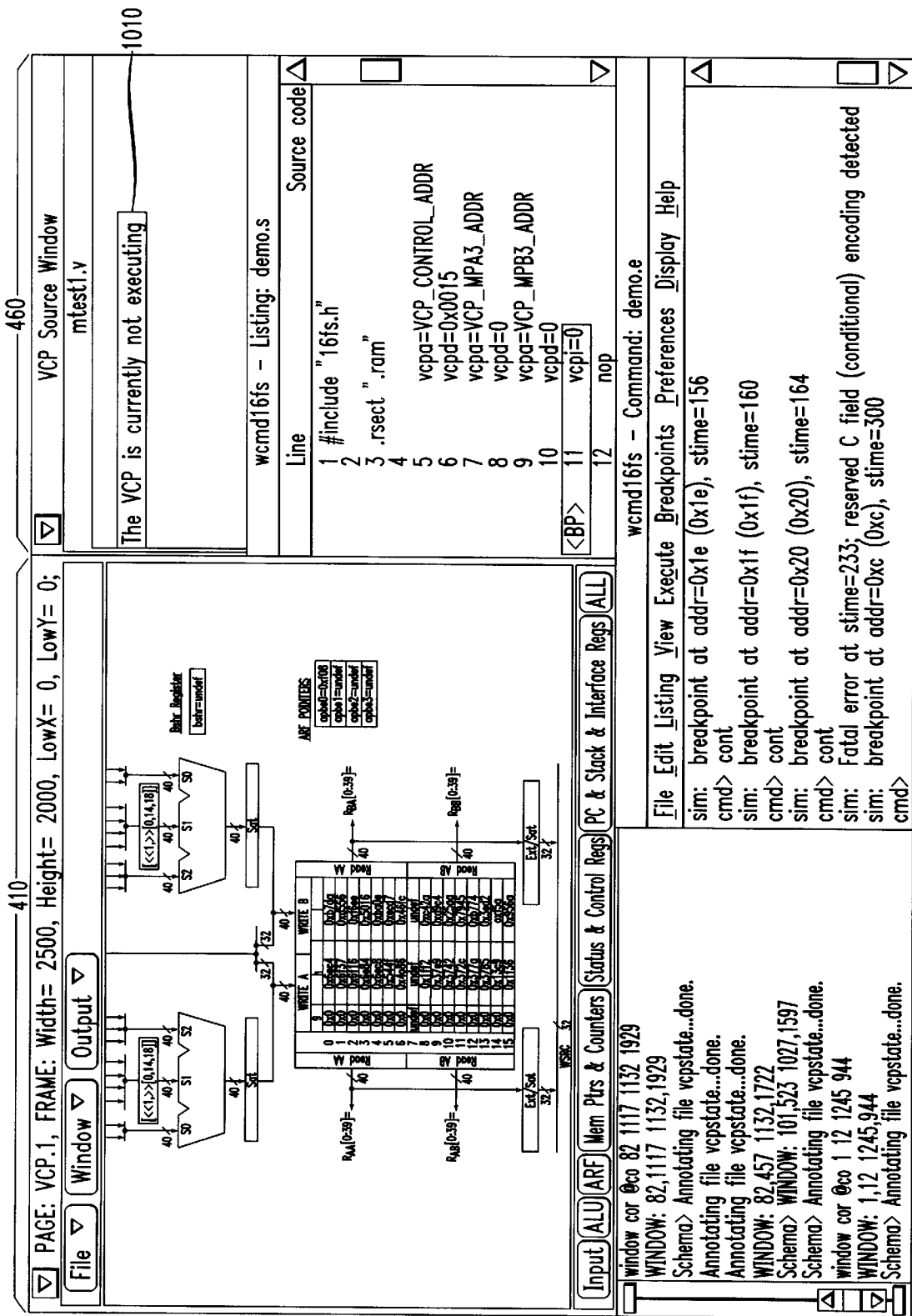


FIG. 11



# SYSTEM AND METHOD FOR DEBUGGING DIGITAL SIGNAL PROCESSOR SOFTWARE WITH AN ARCHITECTURAL VIEW AND GENERAL PURPOSE COMPUTER EMPLOYING THE SAME

## TECHNICAL FIELD OF THE INVENTION

The present invention is directed, in general, to computing systems and, more specifically, to a system and method for simulating the software that is to control a digital signal processor ("DSP") and a general purpose computer employing either the system or the method for simulating.

## BACKGROUND OF THE INVENTION

Digital signal processing has become a mainstream technology employed by designers at the integrated circuit, board and system design levels. The technology is a core ingredient in many technology-related fields including the telecommunications and computer industries. For instance, mobile phones, modems and a myriad of related products depend on digital signal processors ("DSPs") for their basic functionality. DSPs are also effectively employed in audio, video and multimedia peripherals, disk drives and image processing systems for medical and scientific applications. As further developments in the DSP technology occur, DSPS will be advantageously employed in multitude of other applications.

First generation DSPs were highly specialized processors designed for executing various processes on digital signals such as digital computation and conversion based upon retrieval, filtering, equalization, removal of noise or echoes, modulation, Fourier transformations, extraction of the characteristic parameter of a signal, prediction and picture emphasis. The designs were characterized by pipelining arrangements for high speed arithmetic, memory, access and input/output ("I/O") functions. In addition to the capabilities previously mentioned, the DSPs now include general purpose hardware capabilities that, in combination with application programs, provide enhanced system functionality such as real-time control, communication with other processors, memory management and I/O management. For instance, a DSP may operate as a co-processor together with a general purpose microprocessor as the host.

In short, DSPs are basically microcomputers designed to perform specific signal processing tasks under software control. Thus, a high degree of the functionality of the DSP is entrusted to the applications software program. Consequently, an applications developer or programmer (in general, "user") is presented with several challenges regarding the analysis and debugging of the software. The challenges are due to the fact that the user is not only designing a signal processing system, but is also writing software.

Software debugging is not a trivial process. The process includes several tasks, such as ensuring that the software cycles correctly on the DSP. Additionally, the signal processing of the DSP, via an ad hoc and empirical measure of performance, must be periodically verified. Several errors may be detected as a result of the software debugging and signal analysis process including a fundamental error in the signal processing algorithm, a data error in the coefficients, parameters or constants or a software bug. An analysis of signals entering and exiting the DSP are important in determining the source of any error in the operation of the DSP.

Moreover, a conventional DSP software debugging process provides a view of the state of the registers and memory of the machine as the operation of the DSP is emulated and

a software applications program (i.e., the software under test) is simulated thereon. The process of simulating the applications program on the emulated DSP generally occurs on a computer system such as a personal computer.

Conventionally, the values of the registers and memory of the machine are textually displayed on the display of the personal computer such that a user may examine the values as the program is being operated on the emulated DSP. In a sense, the debugging is a brute force process where the user pictures in his mind the state of the machine that correlates with the operation of the machine at a particular stage and throughout the application of the program. The user, then, compares the picture in mind with the textual information provided and makes adjustments to the code, if necessary.

Of course, this brute force evaluation of the state of the machine in connection with the application program running on the emulated DSP is a very arduous task for the user. Additionally, a DSP provides a high level parallel processing capability thereby adding to the complexity of the debugging process. So, not only must the user be concerned with the states of the machine, but the throughput of the machine must be concurrently evaluated. In other words, the user must simultaneously evaluate if the applications program is driving the machine to its fullest potential to achieve the highest level of efficiency.

Accordingly, what is needed in the art is a more efficient way of debugging DSP software, such that development of the program logic is faster and more reliable. Further, what is needed in the art is a way of presenting information about the DSP logic to a user to allow the user to utilize DSP resources most effectively.

## SUMMARY OF THE INVENTION

To address the above-discussed deficiencies of the prior art, the present invention provides a system and method, operable on a general purpose computer, for debugging software that is to control a DSP and a general purpose computer employing either the system or the method. The present invention is employable with either a real DSP or an emulated DSP. When the DSP is emulated, the system includes: (1) architectural display circuitry that displays an architecture of a particular DSP in a window on a display of the general purpose computer, the architecture including a graphical device layout and at least one field corresponding to a register of the DSP and (2) software simulation circuitry that employs a processor of the general purpose computer to simulate operation of DSP software and emulate operation of the particular DSP to cause the particular DSP to change states over time, the architectural display circuitry updating the at least one field to reflect changes in the states, the architectural display circuitry and the software simulation circuitry cooperating to allow a user to debug the software by visually inspecting the graphical device layout and the at least one field.

The present invention therefore introduces, in one embodiment thereof, a combined DSP hardware emulator and software simulator that provides a user an enhanced understanding of DSP states, allowing the user to assess whether the software is functioning as intended. The system is designed for flexibility: the particular DSP may be changed without having to provide additional hardware, and the field(s) corresponding to a register (or multiple registers) of the DSP are preferably user-configurable. In the embodiment of the present invention that operates with a real DSP, detailed information concerning the states can be correlated with the graphical device layout to afford the user a comprehensive understanding of the execution of the DSP software.

Although the terms are sometimes used interchangeably in the art, "emulation" and "simulation" are given different meanings for purposes of the present invention. "Emulation" is an imitation or modeling of the hardware, while "simulation" is execution of software in the emulated hardware.

Alternatively, the system of the present invention may include an interface between a processor of the general purpose computer and a real (non-emulated) DSP. In this alternative embodiment, the general purpose computer causes the DSP software to execute within the real DSP, in turn causing the real DSP to change states over time. The general purpose computer monitors the states allowing the architectural display circuitry to update at least one field to reflect changes in the states.

In an alternative embodiment of the present invention, the system further comprises an architecture database, storable on a storage device of the general purpose computer, that contains a plurality of user-selectable architectures corresponding to a plurality of DSPs, the system thereby allowing the user to select the particular DSP from the database. The present invention therefore can accommodate multiple DSP architectures that the user can recall at will. Additionally, the present invention allows one or more of the user-selectable architectures correspond to DSPs that do not even exist. This gives the user the unique advantage of being able to develop DSP software for a DSP that has not yet been produced. Consequently, introduction of DSP hardware and software can coincide.

In an alternative embodiment of the present invention, the system further comprises source software display circuitry that displays a source code representation of the DSP software in a further window on the display of the general purpose computer to allow the user to debug the software by visually inspecting the graphical device layout, the at least one field and the source code representation. Those skilled in the art are familiar with source code representations employed in more familiar interactive debuggers. The present invention can take advantage of such representations to aid the user in debugging DSP software.

In an alternative embodiment of the present invention, the system further comprises object software display circuitry that displays an object code representation of the DSP software in a further window on the display of the general purpose computer to allow the user to debug the software by visually inspecting the graphical device layout, the at least one field and the object code representation. Likewise, those skilled in the art are familiar with object code representations employed in more familiar interactive debuggers. As with source code representations, the present invention can take advantage of object code representations to aid the user in debugging DSP software.

In an alternative embodiment of the present invention, the system further comprises breakpoint circuitry that allows the user to establish at least one breakpoint for interrupting the operation of the DSP software. Breakpoints allow the user to predefine pausing points, permitting the user to examine DSP states at the breakpoints. Those skilled in the art will recognize, however, that the broad scope of the present invention is not so limited to including breakpoint circuitry.

In an alternative embodiment of the present invention, the architectural display circuitry allows the user to specify a level of detail regarding the graphical device layout to be displayed in the window. The level of detail may be had by zooming in or out or may be had by displaying more or less DSP architecture detail, depending upon the user's wishes.

In an alternative embodiment of the present invention, the architectural display circuitry and the software simulation

circuitry are embodied in a sequence of instructions executable on the processor of the general purpose computer. Alternatively, the present invention may be embodied in dedicated or hardwired discrete or integrated circuitry.

The foregoing has outlined, rather broadly, preferred and alternative features of the present invention so that those skilled in the art may better understand the detailed description of the invention that follows. Additional features of the invention will be described hereinafter that form the subject of the claims of the invention. Those skilled in the art should appreciate that they can readily use the disclosed conception and specific embodiment as a basis for designing or modifying other structures for carrying out the same purposes of the present invention. Those skilled in the art should also realize that such equivalent constructions do not depart from the spirit and scope of the invention in its broadest form.

#### BRIEF DESCRIPTION OF THE DRAWINGS

For a more complete understanding of the present invention, reference is now made to the following descriptions taken in conjunction with the accompanying drawings, in which:

FIG. 1 illustrates an isometric view of an embodiment of a programmable general purpose computer constructed according to the principles of the present invention;

FIG. 2 illustrates a block diagram of an embodiment of the processor associated with the general purpose computer of FIG. 1;

FIG. 3 illustrates an embodiment of a method, performable on a general purpose computer, of simulating software that is to control a DSP;

FIG. 4 illustrates an embodiment of a display for a general purpose computer constructed according to the principles of the present invention; and

FIGS. 5 to 11 illustrate a simulation of DSP software that is to control a particular DSP according to the principles of the present invention.

#### DETAILED DESCRIPTION

Referring initially to FIG. 1, illustrated is an isometric view of an embodiment of a programmable general purpose computer **100** constructed according to the principles of the present invention. The computer **100** is presented as a general purpose computer capable of storing and executing a sequence of instructions to yield a combined DSP hardware emulator and DSP software simulator that provides a user an enhanced understanding of DSP states, allowing the user to assess whether the software is functioning as intended. Since the present invention is not limited to application in a general purpose computer environment, however, FIG. 1 is presented only for illustrative purposes.

The computer **100** includes a monitor or display **110**, a chassis **120** and a keyboard **130**. The monitor **110** and the keyboard **130** cooperate to allow communication (e.g., via a graphical user interface, or "GUI") between the computer **100** and the user. Alternatively, the monitor **110** and keyboard **130** may be replaced by other conventional output input devices, respectively. The chassis **120** includes both a floppy disk drive **140** and hard disk drive **145**. The floppy disk drive **140** is employed to receive, read and write to removable disks; the hard disk drive **145** is employed for fast access storage and retrieval, typically to a nonremovable disk. The floppy disk drive **140** may be replaced by or combined with other conventional structures to receive and transmit data and instructions, including without limitation,



tape and compact disc drives, telephony systems and devices (including videophone, paging and facsimile technologies), and serial and parallel ports.

The chassis **120** is illustrated having a cut-away portion that includes a battery **150**, clock **160**, processor **170** (e.g., Sun Microsystems Sparc 20 as manufactured by Sun Microsystems, Inc. of Mountain View, Calif.) and memory **180**. Although the computer **100** is illustrated having a single processor **170**, hard disk drive **145** and memory **180**, the computer **100** may be equipped with a plurality of processors and peripheral devices.

It should be noted that any conventional computer system having at least one processor that is suitable to function as a general purpose computer may replace, or be used in conjunction with, the computer **100**, including, without limitation: hand-held, laptop/notebook, mini, mainframe and supercomputers, including RISC and parallel processing architectures, as well as within computer system/network combinations. Alternative computer system embodiments may be firmware-or hardware-based.

Before undertaking more detailed discussions of advantageous embodiments of the present invention, the meaning of the following terms and phrases should be understood: the term "or" is inclusive, meaning and/or; the terms "include," "includes" or "including" mean inclusion without limitation; the phrase "associated with" and derivatives thereof may mean to include within, interconnect with, contain, be contained within, connect to or with, couple to or with, be communicable with, juxtapose, cooperate with, interleave, be a property of, be bound to or with, have, have a property of, or the like; and the phrase "memory map" and derivatives thereof may mean a method by which a computer translates between logical and physical address space, and vice versa.

Turning now to FIG. 2, illustrated is a block diagram of an embodiment of the processor **170** associated with the general purpose computer **100** of FIG. 1. The processor **170** is coupled to the memory **180** by a data bus **210**. The memory **180** generally stores data and instructions that the processor **170** uses to execute the functions necessary to operate the computer **100**. The memory **180** may be any conventional memory storage device. The processor **170** includes a control unit **220**, arithmetic logic unit ("ALU") **230** and local memory **240** (e.g., stackable cache or a plurality of registers). The control unit **220** fetches the instructions from memory **180**. The ALU **230**, in turn, performs a plurality of operations, including addition and boolean AND, necessary to carry out the instructions fetched from the memory **180**. The local memory **240** provides a local high speed storage location for storing temporary results and control information generated and employed by the ALU **230**.

In alternate advantageous embodiments, the processor **170** may, in whole or in part, be replaced by or combined with any suitable processing configuration, including multi and parallel processing configurations, programmable logic devices, such as programmable array logic ("PALs") and programmable logic arrays ("PLAs"), digital signal processors ("DSPs"), field programmable gate arrays ("FPGAs"), application specific integrated circuits ("ASICs"), large scale integrated circuits ("LSIs"), very large scale integrated circuits ("VLSIs") or the like, to form the various types of circuitry, controllers and systems described and claimed herein.

It should be noted also that while the processor **170** includes the bus configuration as illustrated, alternate configurations are well within the broad scope of the present

invention. Furthermore, conventional computer system architecture is more fully discussed in *The Indispensable PC Hardware Book*, by Hans-Peter Messmer, Addison Wesley (2nd ed. 1995) and *Computer Organization and Architecture*, by William Stallings, MacMillan Publishing Co. (3rd ed. 1993); conventional computer, or communications, network design is more fully discussed in *Data Network Design*, by Darren L. Spohn, McGraw-Hill, Inc. (1993) and conventional data communications is more fully discussed in *Voice and Data Communications Handbook*, by Bud Bates and Donald Gregory, McGraw-Hill, Inc. (1996), *Data Communications Principles*, by R. D. Gitlin, J. F. Hayes and S. B. Weinstein, Plenum Press (1992) and *The Irwin Handbook of Telecommunications*, by James Harry Green, Irwin Professional Publishing (2nd ed. 1992). Each of the foregoing publications is incorporated herein by reference for all purposes.

In one embodiment of the present invention, the system for simulating software that is to control a DSP operates as follows. The memory **180** stores a plurality of user-selectable architectures corresponding to a plurality of DSPs in a database associated therewith. The display **110**, coupled to the memory **180**, displays an architecture of a particular DSP selected from the database in a window on the display **110**; the architecture includes a graphical device layout and at least one field corresponding to a register of the DSP. The processor **170**, coupled to the display **110**, simulates operation of the DSP software and emulates operation of the particular DSP to cause the particular DSP to change states over time. The processor **170** controls the display **110** to update the at least one field to reflect changes in the states to allow the user to debug the software by visually inspecting the graphical device layout and the at least one field.

Alternatively, the system of the present invention may include an interface **195** and a real (non-emulated) DSP **190** associated with the computer **100**. In this alternative, the processor **170** causes the DSP software to execute within the real DSP **190**, in turn causing the real DSP **190** to change states over time. The processor **170** controls the display **110** to update at least one field of the real DSP **190** to reflect changes in the states to allow the user to debug the software by visually inspecting the graphical device layout and the at least one field.

Turning now to FIG. 3, illustrated is an embodiment of a method, performable on a general purpose computer (such as that illustrated in FIG. 1), of simulating software that is to control a DSP. The method commences at a start step **310**. The software is loaded into the computer in a load DSP software step **320**. An architecture of a particular DSP is displayed in a window on a display of the computer during a display architecture step **330**. The architecture includes a graphical device layout and at least one field corresponding to a register of the DSP. A source code representation of the DSP software is displayed in a further window on the display of the computer during a display source code step **340**. The source code representation allows the user to debug the software by visually inspecting the graphical device layout, the at least one field and the source code representation.

Concurrently, an object code representation of the DSP software is displayed in a further window on the display of the computer during a display object code step **350**. The object code representation allows the user to debug the software by visually inspecting the graphical device layout, the at least one field and the object code representation. In an establish breakpoint step **360**, the user establishes at least one breakpoint for interrupting the operation of the DSP

software. A processor of the computer is employed to simulate operation of the DSP software and emulate operation of the particular DSP to cause the particular DSP to change states over time during a simulate operation of DSP software step 370. In the present embodiment, at each breakpoint the at least one field is updated to reflect changes in the states to allow a user to debug the software by visually inspecting the graphical device layout and the at least one field during an update field step 380. The previously described steps are repeated until the software is meticulously debugged. The method concludes at an end step 390.

Turning now to FIG. 4, illustrated is an embodiment of a display 400 for a general purpose computer constructed according to the principles of the present invention. The display 400 includes an architectural window 410 that displays an architectural view of a particular DSP. The architecture view includes a graphical device layout 420 (sometimes referred to as a "DSP pipeline") and at least one field corresponding to a register 430 of the DSP. The particular DSP is set in an initial state and consequently the parameters and registers in the architectural view are undefined. The display 400 also includes a debugging statement window 440 that displays the debugging statements corresponding to the architectural view. The display 400 also includes a debugging textual interface window 450 that displays an interface to the tools for debugging the DSP.

The display 400 further includes a source window 460 that displays a source code representation of the DSP software to allow the user to debug the software by visually inspecting the graphical device layout 420, the at least one field 430 and the source code representation. The display still further includes another source code window 470 for a second and controlling DSP that is not illustrated in the architectural window 410. Of course, the display 400 may also include an object window (not shown in the illustrated embodiment) that displays an object code representation of the DSP software to further assist the user to debug the software.

In a representative embodiment of the present invention, the system for simulating software that is to control the DSP operates as follows. Architectural display circuitry displays the architecture of the particular DSP in the architectural window 410 including the graphical device layout 420 and the at least one field corresponding to a register 430 of the DSP. The system includes software simulation circuitry that employs a processor of the computer to simulate operation of the DSP software and emulate operation of the particular DSP to cause the particular DSP to change states over time. The architectural display circuitry updates the at least one field 430 to reflect changes in the states. The architectural display circuitry and the software simulation circuitry cooperate to allow the user to debug the software by visually inspecting the graphical device layout 420 and the at least one field 430. The architectural display circuitry also allows the user to specify a level of detail regarding the graphical device layout 420 to be displayed. The system further includes an architecture database, storable on a storage device or memory of the computer, that contains a plurality of user-selectable architectures corresponding to a plurality of DSPs; the system thereby allows the user to select the particular DSP from the database. The system further includes source software display circuitry that displays the source code representation of the DSP software as discussed above. The system still further includes breakpoint circuitry that allows the user to establish at least one breakpoint for interrupting the operation of the DSP software.

A processor and memory of the general purpose computer cooperate to form the architectural display circuitry, soft-

ware simulation circuitry, software execution circuitry, source software display circuitry, object software display circuitry, breakpoint circuitry and architectural database as described above. The architectural display circuitry, software simulation circuitry, software execution circuitry, source software display circuitry, object software display circuitry and breakpoint circuitry may also be embodied in a sequence of instructions executable on the processor of the general purpose computer. Thus, the present invention may be embodied in software, dedicated or hardwired discrete or integrated circuitry, or combinations thereof.

Turning sequentially to FIGS. 5 to 11, illustrated is a simulation of the DSP software that is to control a particular DSP according to the principles of the present invention. Again, the present invention introduces a combined DSP hardware emulator and software simulator that provides a user an enhanced understanding of DSP states, thereby allowing the user to assess whether the software is functioning as intended. The simulation of the DSP software will hereinafter be described employing the display 400 illustrated and described with respect to FIG. 4. Table I below lists a source code representation of the DSP software simulated on the particular DSP. The source code representation is displayed in the source window 460.

TABLE I

SOURCE CODE	COMMENTS
vda.1=*mpa3++2	load vda.1 registers with value of memory at address mpa3; increment mpa3 by 2
vc0=vda.1	initialize counter vc0 to vda.1
vapbe0=0x0f00	initialize pointer, base, end registers
alua va0=0, alub va1=0	initialize accumulators 0 & 1 to 0
alua va2=0, alub va3=0	initialize accumulators 2 & 3 to 0
alua va4=0, alub va5=0	initialize accumulators 4 & 5 to 0
alua va6=0, alub va7=0	initialize accumulators 6 & 7 to 0
alua va8=0, alub va9=0	initialize accumulators 8 & 9 to 0
alua va10=0, alub va11=0	initialize accumulators 10 & 11 to 0
alua va12=0, alub va13=0	initialize accumulators 0 & 12 to 13
alua va14=0, alub va15=0	initialize accumulators 14 & 15 to 0
vda=*mpa3++2, vdb=*mpb3++2	load data from memory into vda, vdb
xa=vda.h, ya.0=vdb.h, xb=vda.1, yb.0=vdb.1, vda=*mpa3++2, vdb=*mpb3++2	load x, y registers on both sides while also loading the memory registers
pa=xa*ya.h, pb=xb*yb.h	compute first product
loop: if vc0ge goto loop: alua *vap0+=*vap0+pa+pb, pa=xa*ya.h, pb=xb*yb.h, xa=vda.h, ya.0=vdb.h, xb=vda.1, yb.0=vdb.1, vda=*mpa3++2, vdb=*mpb3++2	all at the same time, accumulate a sum in arf, calculate next product, load x and y registers, and load memory registers in both pipes
end: alua *vap0+=*vap0+pa+pb	calculate last sum

With specific reference to FIG. 5, the system (as described with respect to FIG. 4) has performed one instruction and a subsequent instruction (denoted by a first highlighted instruction 510) is ready to be executed. As represented in a higher resolution illustration of the architectural window 410, a value (indicated in a memory block 520 of the DSP pipeline) is now displayed for "vda.1" resulting from the execution of the initial instruction. A breakpoint (denoted by

a first highlighted term **530**) for interrupting the operation of the DSP software is illustrated in the source window **470**.

With specific reference to FIG. 6, the system has performed all of the instructions up to the next ready instruction (denoted by a second highlighted instruction **610**). As represented in another higher resolution illustration of the architectural window **410**, all of the corresponding states of the registers in the DSP pipeline have been updated including the accumulator registers **620** being initialized to zero.

With specific reference to FIG. 7, the system has performed all of the instructions up to the next ready instruction (denoted by a third highlighted instruction **710**). The previous instruction loaded the vda and vdb registers from the memory. The third highlighted instruction **710** will load the x and y registers in conjunction with loading the vda and vdb registers once again. The instructions are performed in parallel. As represented in another higher resolution illustration of the architectural window **410**, all of the corresponding states of the registers in the DSP pipeline have been updated to represent the latest instructions.

With specific reference to FIG. 8, the system has performed all of the instructions up to the next ready instruction (denoted by a fourth highlighted instruction **810**). The previous instruction loaded the xa, ya, xb, yb, vda and vdb registers from the memory. The fourth highlighted instruction **810** will compute the first products of the respective registers. As represented in another higher resolution illustration of the architectural window **410**, all of the corresponding states of the registers in the DSP pipeline have been updated to represent the latest instructions.

With specific reference to FIG. 9, the system has performed all of the instructions up to the next ready instruction (denoted by a fifth highlighted instruction **910**). The previous instruction computed the first products introduced with respect to FIG. 8. As represented in another higher resolution illustration of the architectural window **410**, all of the corresponding states of the registers in the DSP pipeline have been updated to represent the latest instructions.

With specific reference to FIG. 10, the system has performed all of the instructions up to and including the next ready instruction (denoted by a sixth highlighted instruction **1010**). The sixth highlighted instruction **1010** has added the previous values of pa and pb and placed the result in the va0 register, computed new products in the first and second column of the DSP pipeline, loaded both sets of the x and y registers from the previous values of vda and vdb, and read the next value in memory into the vda and vdb registers. As represented in another higher resolution illustration of the architectural window **410**, all of the corresponding states of the registers in the DSP pipeline have been updated to represent the latest instructions.

With specific reference to FIG. 11, the system has performed all of the instructions and the program is currently not executing (denoted by a seventh highlighted instruction **1110**). The final instructions of the program computed additional sums and stored the values into the accumulators of the DSP pipeline. As represented in another higher resolution illustration of the architectural window **410**, all of the corresponding states of the registers in the DSP pipeline have been updated to represent the remaining instructions.

The previously described simulation of the DSP software to control a particular DSP is presented for illustrative purposes only. Any system capable of allowing a user to debug the DSP software by visually inspecting the graphical device layout and at least one field thereof is well within the broad scope of the present invention. Furthermore, the present invention may operate on more than one DSP at the same time.

For a better understanding of DSPs see *VLSI Digital Signal Processors*, by Vijay K. Madiseti, Butterworth-Heinemann (1995). The aforementioned reference is herein incorporated by reference.

Although the present invention has been described in detail, those skilled in the art should understand that they can make various changes, substitutions and alterations herein without departing from the spirit and scope of the invention in its broadest form.

What is claimed is:

1. A system, operable on a general purpose computer, for simulating software that is to control a digital signal processor (DSP), comprising:

architectural display circuitry that displays an architecture of a particular DSP in a window on a display of said general purpose computer, said architecture including a graphical device layout and at least one field corresponding to a register of said DSP; and

software simulation circuitry that employs a processor of said general purpose computer to simulate operation of DSP software and emulate operation of said particular DSP to cause said particular DSP to change states over time, said architectural display circuitry updating said at least one field to reflect changes in said states, said architectural display circuitry and said software simulation circuitry cooperating to allow a user to debug said software by visually inspecting said graphical device layout and said at least one field.

2. The system as recited in claim 1 further comprising an architecture database, storable on a storage device of said general purpose computer, that contains a plurality of user-selectable architectures corresponding to a plurality of DSPs, said system thereby allowing said user to select said particular DSP from said database.

3. The system as recited in claim 1 further comprising source software display circuitry that displays a source code representation of said DSP software in a separate window on said display of said general purpose computer to allow said user to debug said software by visually inspecting said graphical device layout, said at least one field and said source code representation.

4. The system as recited in claim 1 further comprising object software display circuitry that displays an object code representation of said DSP software in a separate window on said display of said general purpose computer to allow said user to debug said software by visually inspecting said graphical device layout, said at least one field and said object code representation.

5. The system as recited in claim 1 further comprising breakpoint circuitry that allows said user to establish at least one breakpoint for interrupting said operation of said DSP software.

6. The system as recited in claim 1 wherein said architectural display circuitry allows said user to specify a level of detail regarding said graphical device layout to be displayed in said window.

7. The system as recited in claim 1 wherein said architectural display circuitry and said software simulation circuitry are embodied in a sequence of instructions executable on said processor of said general purpose computer.

8. A method, performable on a general purpose computer, of simulating software that is to control a digital signal processor (DSP), comprising the steps of:

displaying an architecture of a particular DSP in a window on a display of said general purpose computer, said architecture including a graphical device layout and at least one field corresponding to a register of said DSP;

## 11

employing a processor of said general purpose computer to simulate operation of DSP software and emulate operation of said particular DSP to cause said particular DSP to change states over time; and

updating said at least one field to reflect changes in said states to allow a user to debug said software by visually inspecting said graphical device layout and said at least one field.

9. The method as recited in claim 8 further comprising the step of containing a plurality of user-selectable architectures corresponding to a plurality of DSPs in a database, said system thereby allowing said user to select said particular DSP from said database.

10. The method as recited in claim 8 further comprising the step of displaying a source code representation of said DSP software in a separate window on said display of said general purpose computer to allow said user to debug said software by visually inspecting said graphical device layout, said at least one field and said source code representation.

11. The method as recited in claim 8 further comprising the step of displaying an object code representation of said DSP software in a separate window on said display of said general purpose computer to allow said user to debug said software by visually inspecting said graphical device layout, said at least one field and said object code representation.

12. The method as recited in claim 8 further comprising the step of allowing said user to establish at least one breakpoint for interrupting said operation of said DSP software.

13. The method as recited in claim 8 wherein said architectural display circuitry allows said user to specify a level of detail regarding said graphical device layout to be displayed in said window.

14. The method as recited in claim 8 wherein said steps of displaying, employing and updating are performed by executing a sequence of instructions on said processor of said general purpose computer.

15. A programmable general purpose computer, comprising:

a memory that stores a plurality of user-selectable architectures corresponding to a plurality of digital signal processors (DSPs) in a database;

a display, coupled to said memory, that displays an architecture of a particular DSP selected from said database in a window on said display, said architecture including a graphical device layout and at least one field corresponding to a register of said DSP;

a processor, coupled to said display, that simulates operation of DSP software and emulates operation of said particular DSP to cause said particular DSP to change states over time, said processor controlling said display to update said at least one field to reflect changes in said states to allow a user to debug said software by visually inspecting said graphical device layout and said at least one field.

16. The computer as recited in claim 15 wherein said user is allowed to select said particular DSP from said database.

17. The computer as recited in claim 15 wherein said processor controls said display to display a source code representation of said DSP software in a separate window on said display of said general purpose computer to allow said user to debug said software by visually inspecting said graphical device layout, said at least one field and said source code representation.

18. The computer as recited in claim 15 wherein said processor controls said display to display an object code representation of said DSP software in a separate window on

## 12

said display of said general purpose computer to allow said user to debug said software by visually inspecting said graphical device layout, said at least one field and said object code representation.

19. The computer as recited in claim 15 wherein said processor allows said user to establish at least one breakpoint for interrupting said operation of said DSP software.

20. The computer as recited in claim 15 wherein said processor allows said user to specify a level of detail regarding said graphical device layout to be displayed in said window.

21. A system, operable on a general purpose computer, for controlling a digital signal processor (DSP), comprising:

architectural display circuitry that displays an architecture of a particular DSP in a window on a display of said general purpose computer, said architecture including a graphical device layout and at least one field corresponding to a register of said DSP; and

software execution circuitry that communicates with a DSP via an interface of said general purpose computer to execute DSP software on said particular DSP to cause said particular DSP to change states over time, said architectural display circuitry updating said at least one field to reflect changes in said states, said architectural display circuitry and said software execution circuitry cooperating to allow a user to debug said software by visually inspecting said graphical device layout and said at least one field.

22. The system as recited in claim 21 further comprising an architecture database, storable on a storage device of said general purpose computer, that contains a plurality of user-selectable architectures corresponding to a plurality of DSPs.

23. The system as recited in claim 21 further comprising source software display circuitry that displays a source code representation of said DSP software in a separate window on said display of said general purpose computer to allow said user to debug said software by visually inspecting said graphical device layout, said at least one field and said source code representation.

24. The system as recited in claim 21 further comprising object software display circuitry that displays an object code representation of said DSP software in a separate window on said display of said general purpose computer to allow said user to debug said software by visually inspecting said graphical device layout, said at least one field and said object code representation.

25. The system as recited in claim 1 further comprising breakpoint circuitry that allows said user to establish at least one breakpoint for interrupting said operation of said DSP software.

26. The system as recited in claim 21 wherein said architectural display circuitry allows said user to specify a level of detail regarding said graphical device layout to be displayed in said window.

27. The system as recited in claim 21 wherein said architectural display circuitry and said software execution circuitry are embodied in a sequence of instructions executable on said processor of said general purpose computer.

28. A method, performable on a general purpose computer, of controlling a digital signal processor (DSP), comprising the steps of:

displaying an architecture of a particular DSP in a window on a display of said general purpose computer, said architecture including a graphical device layout and at least one field corresponding to a register of said DSP; communicating with a DSP via an interface of said general purpose computer to execute DSP software on

## 13

said particular DSP to cause said particular DSP to change states over time; and

updating said at least one field to reflect changes in said states to allow a user to debug said software by visually inspecting said graphical device layout and said at least one field.

29. The method as recited in claim 28 further comprising the step of containing a plurality of user-selectable architectures corresponding to a plurality of DSPs in a database.

30. The method as recited in claim 28 further comprising the step of displaying a source code representation of said DSP software in a separate window on said display of said general purpose computer to allow said user to debug said software by visually inspecting said graphical device layout, said at least one field and said source code representation.

31. The method as recited in claim 28 further comprising the step of displaying an object code representation of said DSP software in a separate window on said display of said general purpose computer to allow said user to debug said software by visually inspecting said graphical device layout, said at least one field and said object code representation.

32. The method as recited in claim 28 further comprising the step of allowing said user to establish at least one breakpoint for interrupting said operation of said DSP software.

33. The method as recited in claim 28 wherein said architectural display circuitry allows said user to specify a level of detail regarding said graphical device layout to be displayed in said window.

34. The method as recited in claim 28 wherein said steps of displaying, communicating and updating are performed by executing a sequence of instructions on said processor of said general purpose computer.

35. A programmable general purpose computer, comprising:

a memory that stores a plurality of user-selectable architectures corresponding to a plurality of digital signal processors (DSPs) in a database;

a display, coupled to said memory, that displays an architecture of a particular DSP selected from said

## 14

database in a window on said display, said architecture including a graphical device layout and at least one field corresponding to a register of said DSP; and

a processor, coupled to said display, that communicates with a DSP via an interface of said general purpose computer to execute DSP software on said particular DSP to cause said particular DSP to change states over time, said processor controlling said display to update said at least one field to reflect changes in said states to allow a user to debug said software by visually inspecting said graphical device layout and said at least one field.

36. The computer as recited in claim 35 wherein said interface circuitry is adapted to be coupled to a selected one of a plurality of DSPs.

37. The computer as recited in claim 35 wherein said processor controls said display to display a source code representation of said DSP software in a separate window on said display of said general purpose computer to allow said user to debug said software by visually inspecting said graphical device layout, said at least one field and said source code representation.

38. The computer as recited in claim 35 wherein said processor controls said display to display an object code representation of said DSP software in a separate window on said display of said general purpose computer to allow said user to debug said software by visually inspecting said graphical device layout, said at least one field and said object code representation.

39. The computer as recited in claim 35 wherein said processor allows said user to establish at least one breakpoint for interrupting said operation of said DSP software.

40. The computer as recited in claim 35 wherein said processor allows said user to specify a level of detail regarding said graphical device layout to be displayed in said window.

\* \* \* \* \*



US 20030110476A1

(19) **United States**(12) **Patent Application Publication** (10) **Pub. No.: US 2003/0110476 A1**  
Aihara (43) **Pub. Date: Jun. 12, 2003**(54) **SOURCE CODE DEBUGGER, DEBUGGING  
METHOD AND DEBUGGING PROGRAM****Publication Classification**(51) **Int. Cl.<sup>7</sup>** ..... **G06F 9/44**(52) **U.S. Cl.** ..... **717/135**(76) **Inventor: Masami Aihara, Tokyo (JP)**

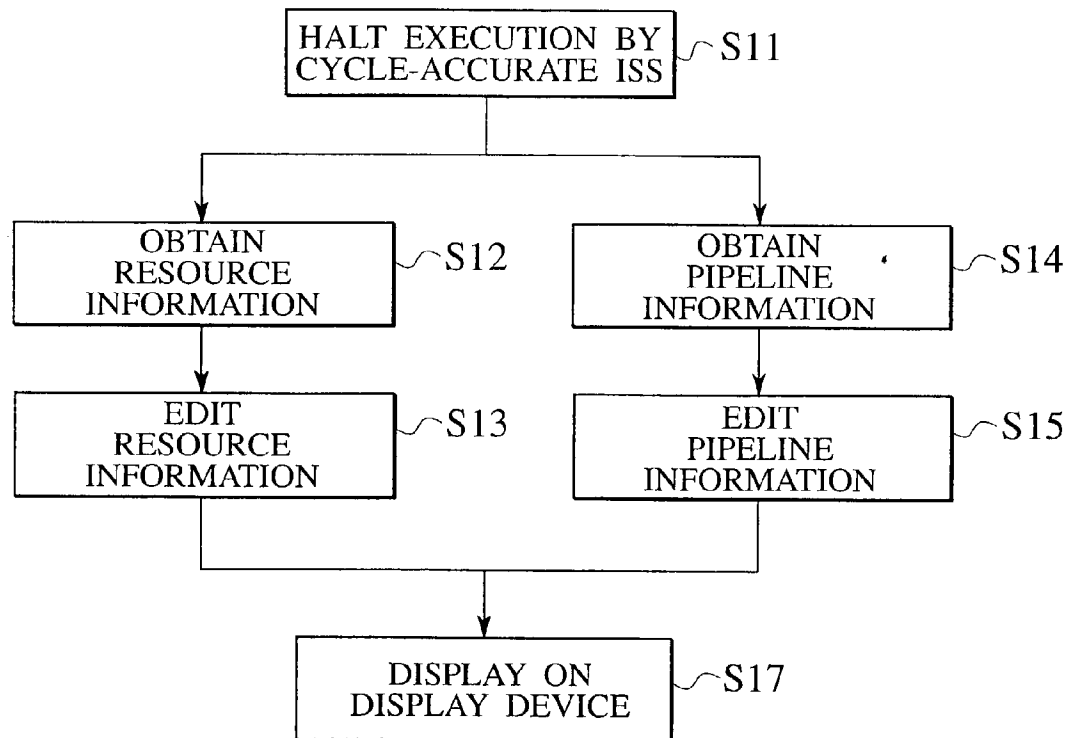
Correspondence Address:

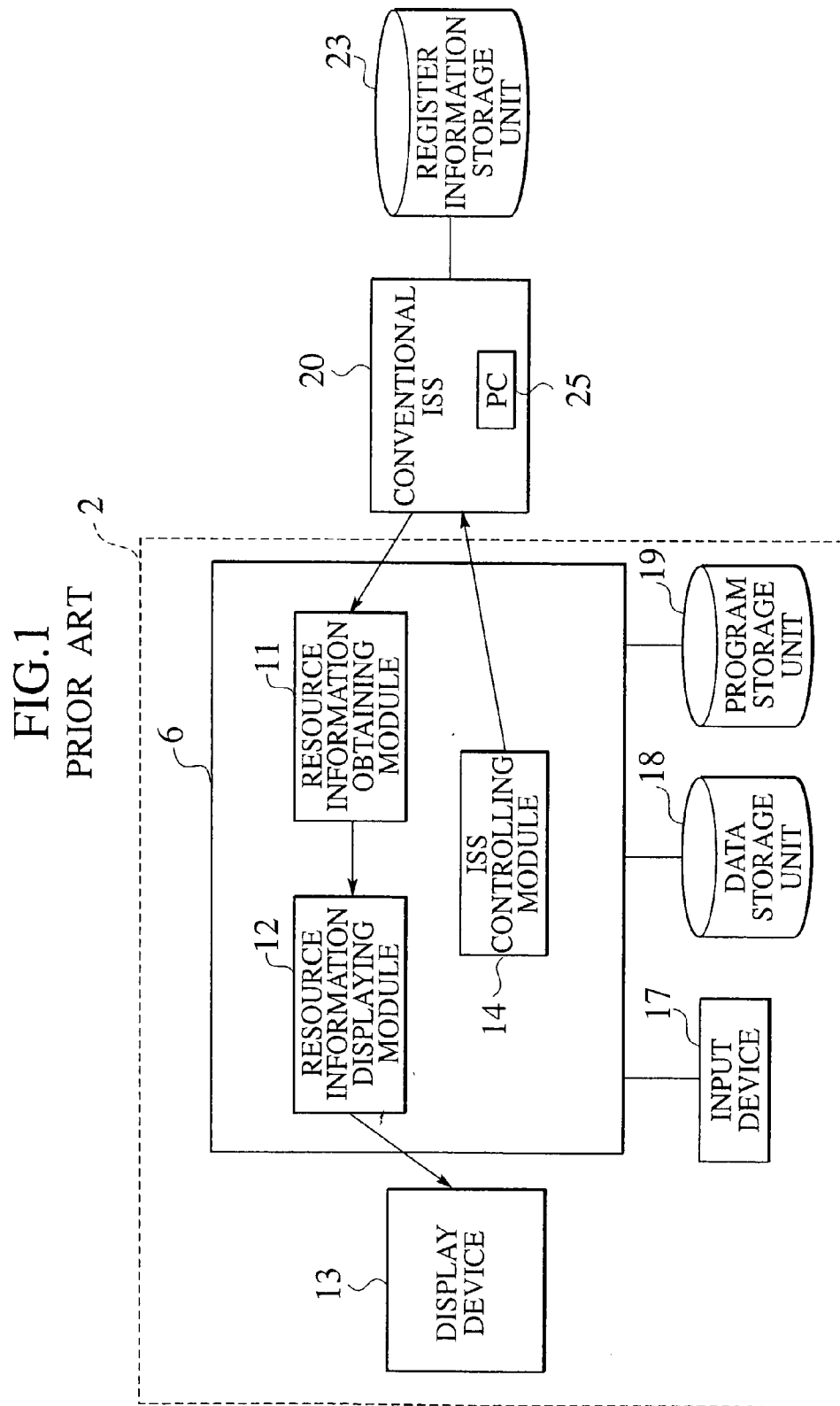
**JOHN S. PRATT, ESQ**  
**KILPATRICK STOCKTON, LLP**  
**1100 PEACHTREE STREET**  
**SUITE 2800**  
**ATLANTA, GA 30309 (US)**(21) **Appl. No.: 10/241,237**(22) **Filed: Sep. 11, 2002**(30) **Foreign Application Priority Data**

Dec. 9, 2001 (JP) ..... P2001-277196

(57) **ABSTRACT**

A source code debugger connected to a cycle-accurate instruction set simulator, comprising: a pipeline information obtaining module configured to obtain address information in execution at respective stages on a pipeline from the cycle-accurate instruction set simulator; a pipeline information displaying module configured to edit the address information of the pipeline obtained by the pipeline information obtaining module together with the progress of processing of the stages; a resource information obtaining module configured to obtain address information of a program in execution together with instruction codes; and a resource information displaying module configured to edit the address information and the instruction codes obtained by the resource information obtaining module.





## FIG.2

### PRIOR ART

Source Code : treset. s		
0x800201f0	:	
0x800201f4	LUI	\$12, 0x8002
0x800201f8	ORI	\$12, 0x0400
➡ 0x80020200	LD	\$3, 0(\$12)
0x80020204	LD	\$4, 4(\$12)
0x80020208	ADD	\$3, \$3, \$4
0x8002020c	:	
0x80020210	:	
0x80020214	:	
0x80020218	:	

## FIG.3

### PRIOR ART

Register Dump		
Name	:	Value
PC	:	0x80020200
\$0	:	0x00000000
\$1	:	0x00000000
\$2	:	0x80020400
\$3	:	0x00000000
:	:	:
:	:	:



## FIG.4 PRIOR ART

Source Code : treset. s		
0x800201f0	:	
0x800201f4	LUI	\$12, 0x8002
0x800201f8	ORI	\$12, 0x0400
0x80020200	LD	\$3, 0(\$12)
➔ 0x80020204	LD	\$4, 4(\$12)
0x80020208	ADD	\$3, \$3, \$4
0x8002020c	:	
0x80020210	:	
0x80020214	:	
0x80020218	:	

## FIG.5 PRIOR ART

Register Dump		
Name	:	Value
PC	:	0x80020204
\$0	:	0x00000000
\$1	:	0x00000000
\$2	:	0x80020400
\$3	:	0x00000064
:	:	:
:	:	:

## FIG.6 PRIOR ART

Source Code : treset. s		
0x800201f0	:	
0x800201f4	LUI	\$12, 0x8002
0x800201f8	ORI	\$12, 0x0400
0x80020200	LD	\$3, 0(\$12)
➔ 0x80020204	LD	\$4, 4(\$12)
0x80020208	ADD	\$3, \$3, \$4
0x8002020c	:	
0x80020210	:	
0x80020214	:	
0x80020218	:	

## FIG.7 PRIOR ART

Register Dump		
Name	:	Value
PC	:	0x80020204
\$0	:	0x00000000
\$1	:	0x00000000
\$2	:	0x80020400
\$3	:	0x00000000
:	:	:
:	:	:

FIG.8  
PRIOR ART

Source Code : treset. s		
0x800201f0	:	
0x800201f4	AND	\$2, \$1, \$2
0x800201f8	LD	\$3, 0(\$12)
➔ 0x80020200	BNE	\$2, \$3, 0x5
0x80020204	NOP	
0x80020208	ADD	\$3, \$3, \$4
0x8002020c	:	
0x80020210	:	
0x80020214	:	
0x80020218	:	

FIG.9  
PRIOR ART

Source Code : treset. s		
0x800201f0	:	
0x800201f4	AND	\$2, \$1, \$2
0x800201f8	LD	\$3, 0(\$12)
0x80020200	BNE	\$2, \$3, 0x5
0x80020204	NOP	
0x80020208	ADD	\$3, \$3, \$4
0x8002020c	:	
0x80020210	:	
➔ 0x80020214	:	
0x80020218	:	

FIG.10  
PRIOR ART

Source Code : treset. s		
0x800201f0	:	
0x800201f4	AND	\$2, \$1, \$2
0x800201f8	LD	\$3, 0(\$12)
0x80020200	BNE	\$2, \$3, 0x5
➔ 0x80020204	NOP	
0x80020208	ADD	\$3, \$3, \$4
0x8002020c	:	
0x80020210	:	
0x80020214	:	
0x80020218	:	

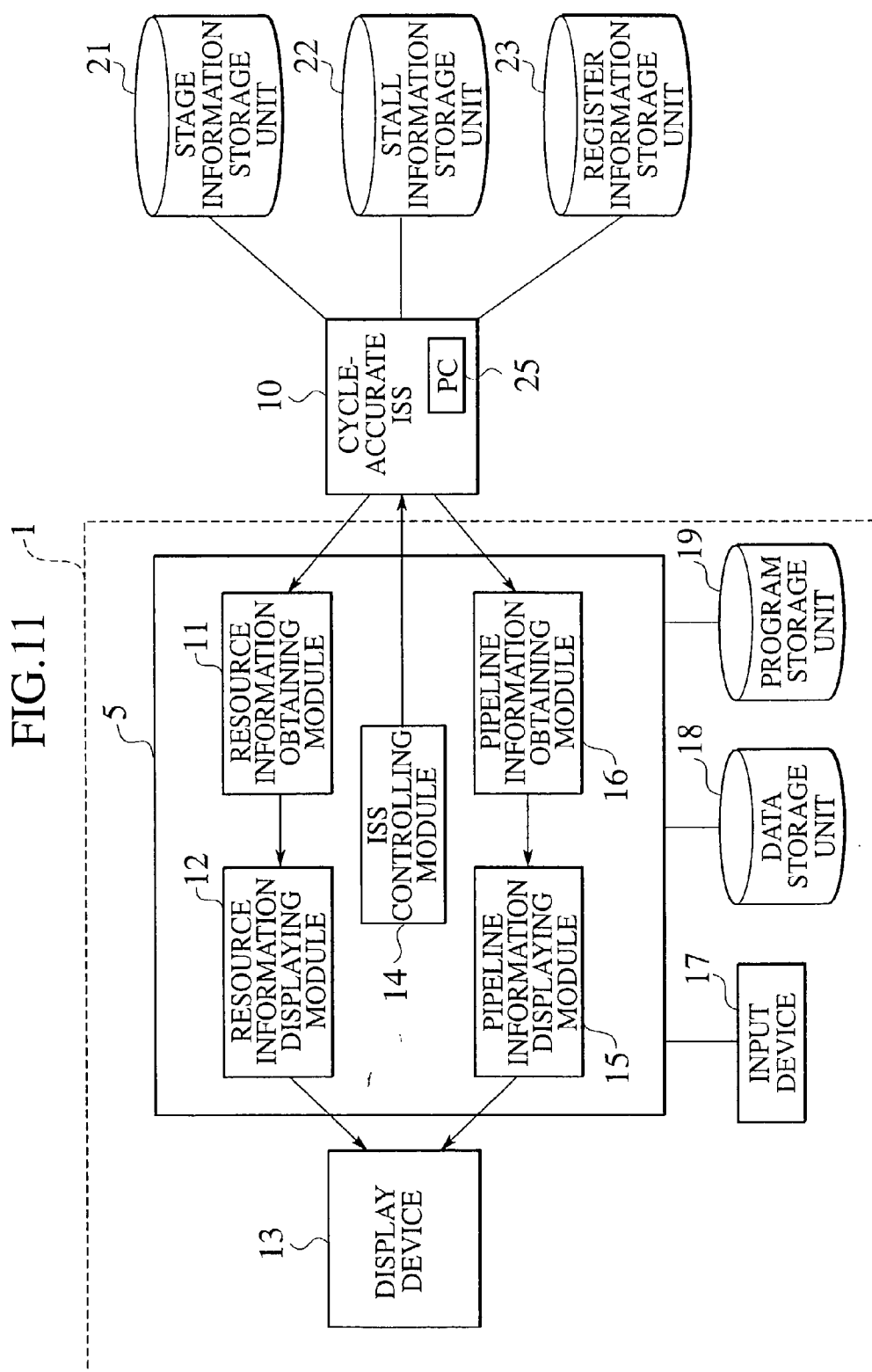


FIG.12

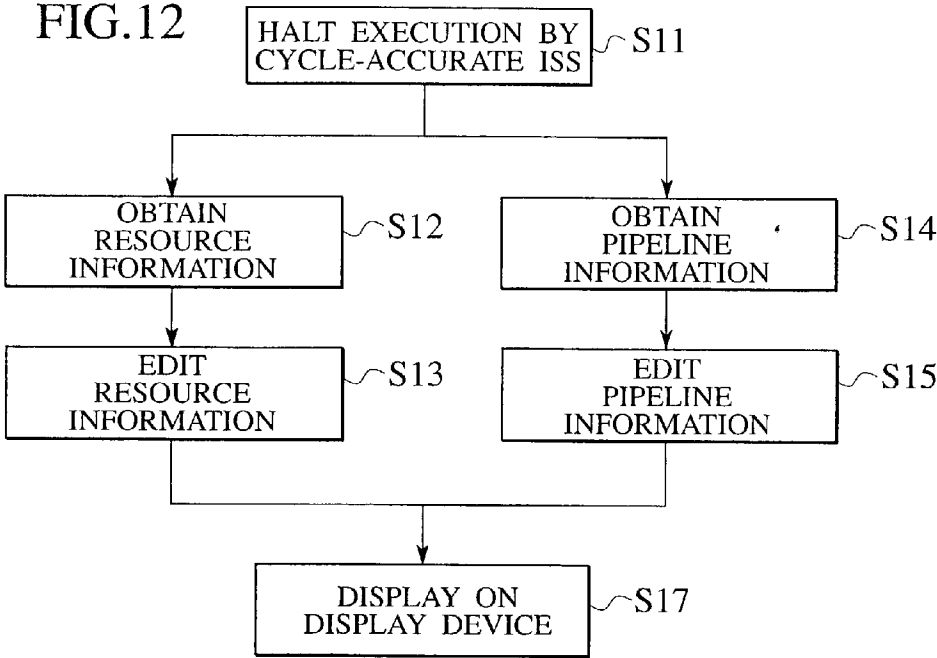


FIG.13A

Source Code : sample. s		
0x800201e0		
0x800201e4	LUI	\$12, 0x8002
0x800201e8	ORI	\$12, 0x0400
0x800201ec	W	LD \$1, 0(\$12)
0x800201f0	M	LD \$2, 4(\$12)
0x800201f4	E	AND \$2, \$1, \$2
0x800201f8	D	LD \$3, 8(\$12)
➔ 0x800201fc	F	BNE \$2, \$3, 0x5
0x80020200		NOP
0x80020204		ADD \$3, \$3, \$4
0x80020208		:
0x8002020c		:
0x80020210		:
0x80020214		:

FIG.13B

Source Code : sample. s	
0x800201e0	
0x800201e4	LUI \$12, 0x8002
0x800201e8	ORI \$12, 0x0400
0x800201ec	LD \$1, 0(\$12)
0x800201f0	LD \$2, 4(\$12)
0x800201f4	AND \$2, \$1, \$2
0x800201f8	LD \$3, 8(\$12)
→ 0x800201fc	BNE \$2, \$3, 0x5
0x80020200	NOP
0x80020204	ADD \$3, \$3, \$4
0x80020208	:
0x8002020c	:
0x80020210	:
0x80020214	:

FIG.14A

Source Code : sample. s	
0x800201e0	
0x800201e4	LUI \$12, 0x8002
0x800201e8	ORI \$12, 0x0400
0x800201ec	<b>W</b> LD \$1, 0(\$12)
0x800201f0	<b>M</b> LD \$2, 4(\$12)
0x800201f4	<b>Es</b> AND \$2, \$1, \$2
0x800201f8	<b>Ds</b> LD \$3, 8(\$12)
→ 0x800201fc	<b>Fs</b> BNE \$2, \$3, 0x5
0x80020200	NOP
0x80020204	ADD \$3, \$3, \$4
0x80020208	:
0x8002020c	:
0x80020210	:
0x80020214	:

FIG.14B

Source Code : sample. s		
0x800201e0		
0x800201e4		LUI \$12, 0x8002
0x800201e8		ORI \$12, 0x0400
0x800201ec	W	LD \$1, 0(\$12)
0x800201f0	M	LD \$2, 4(\$12)
0x800201f4	E	AND \$2, \$1, \$2
0x800201f8	D	LD \$3, 8(\$12)
→ 0x800201fc	F	BNE \$2, \$3, 0x5
0x80020200		NOP
0x80020204		ADD \$3, \$3, \$4
0x80020208		:
0x8002020c		:
0x80020210		:
0x80020214		:

FIG.15

Source Code : sample. s		
0x800201e0		:
0x800201e4		LUI \$12, 0x8002
0x800201e8		ORI \$12, 0x0400
0x800201ec	W	LD \$1, 0(\$12)
0x800201f0	M	LD \$2, 4(\$12)
0x800201f4	Es	AND \$2, \$1, \$2
0x800201f8	E	LD \$3, 8(\$12)
0x800201fc	D	BNE \$2, \$3, 0x5
→ 0x80020200	F	NOP
0x80020204		ADD \$3, \$3, \$4
0x80020208		:
0x8002020c		:
0x80020210		:
0x80020214		:

FIG.16A

Register Dump		
Name : Value		
PC : 0x800201ec		
\$0 : 0x00000000		
\$1	:	0x12345678
\$2	:	0xffffffff
\$3	:	0x00000000
\$4 : 0x00000000		
:	:	:
\$12	:	0x80020400
:	:	:

FIG.16B

Register Dump		
Name : Value		
PC : 0x800201ec		
\$0 : 0x00000000		
W M D	\$1	: 0x12345678
	\$2	: 0xffffffff
	\$3	: 0x00000000
\$4 : 0x00000000		
:	:	:
\$12 : 0x80020400		
:	:	:



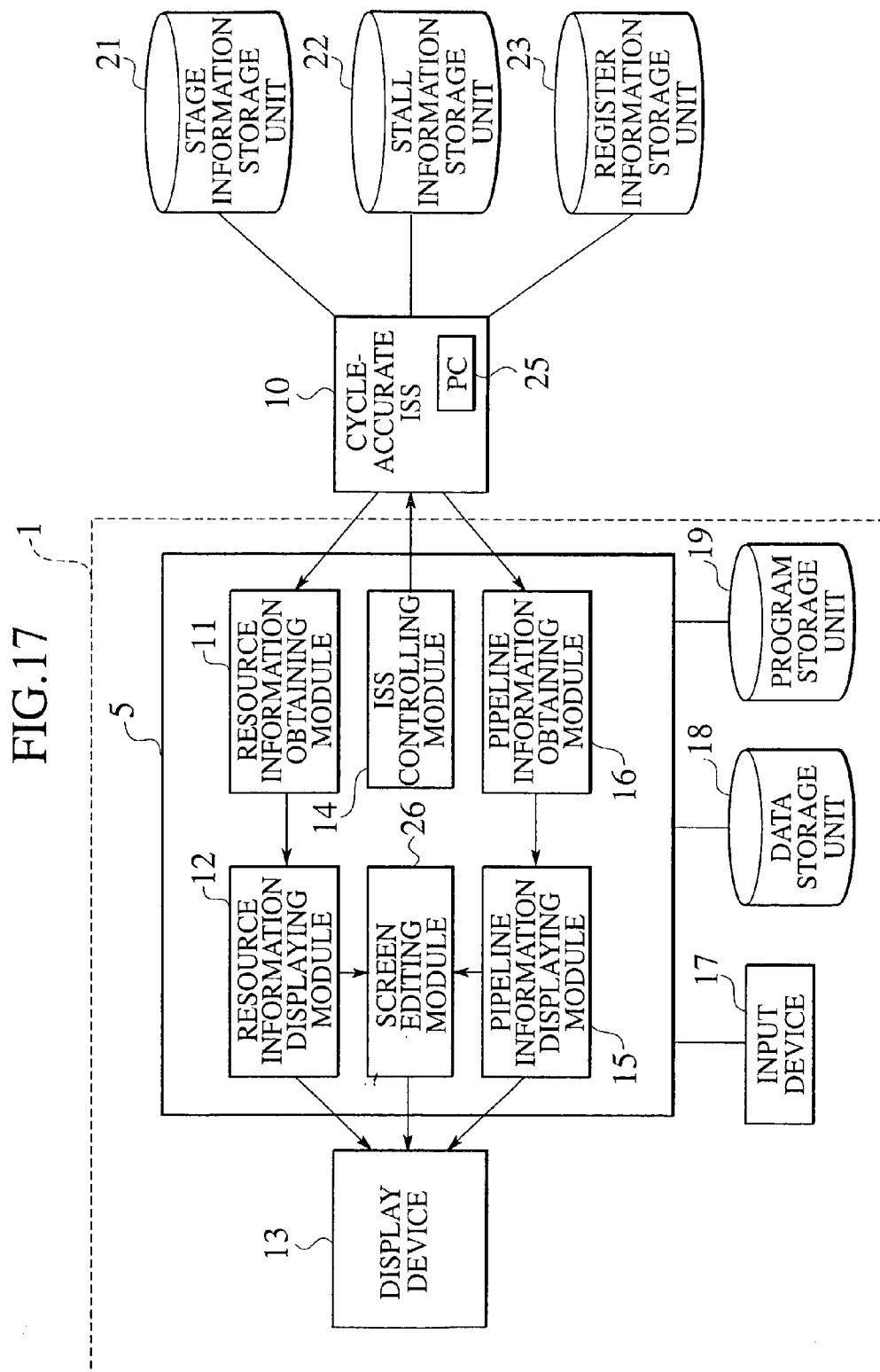


FIG.18

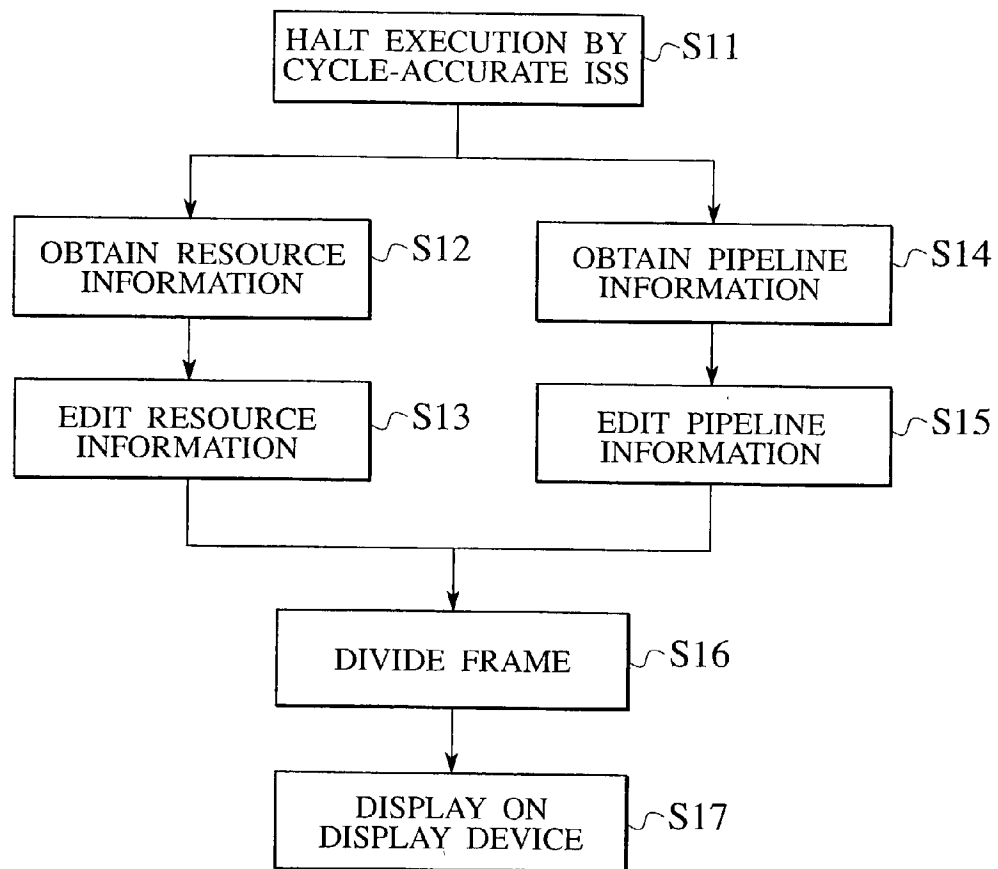


FIG.19

Source Code : sample. s			
0x800201f8		LD	\$3, 8 (\$12)
0x800201fc	W	BNE	\$2, \$3, 0x100
0x80020200	M	NOP	
0x80020204		ADD	\$3, \$3, \$4
0x80020208		:	
0x800202f8		:	
0x800202fc	Es	LDI	\$7, 0x8002
0x80020300	Ds	ORI	\$7, 0x0440
→ 0x80020304	Fs	LS	\$8, 0x (\$7)
0x80020308		:	
0x8002030c		:	
0x80020310		:	
0x80020314		:	
0x80020314		:	

## SOURCE CODE DEBUGGER, DEBUGGING METHOD AND DEBUGGING PROGRAM

### CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This application is based upon and claims the benefit of priority from prior Japanese Patent Application P2001-277196 filed on Sep. 12, 2001; the entire contents of which are incorporated by reference herein.

### BACKGROUND OF THE INVENTION

#### [0002] 1. Field of the Invention

[0003] The present invention relates to source code debuggers for use in software development, more specifically, to a source code debugger, a debugging method and a debugging program, which respond to cycle-accurate ISS taking account of pipeline processing used in hardware-software co-simulation environment.

#### [0004] 2. Description of the Related Art

[0005] Hardware-software co-simulation environment is becoming widespread as an environment for improving efficiency in the development of system LSIs including microprocessors. The hardware-software co-simulation environment is constructed by the integration of a logic simulator, being a conventional hardware development tool, and a processor instruction set simulator (ISS), being a software development tool.

[0006] A conventionally used ISS (hereinafter referred to as a "conventional ISS") is designed to simulate an operation of a processor by each instruction. Meanwhile, the conventional ISS does not consider pipeline stages even if the processor adopts a pipeline processing system.

[0007] In general, a source code debugger is used for controlling the execution of and displaying results from the conventional ISS (such a source code debugger is hereinafter referred to as a "conventional source code debugger"). Famous source code debuggers includes "MULTI" by Green Hills Software, "XRAY" by Mentor Graphics, "GDB" by Red Hat, and the like. FIG. 1 shows one example of a constitution of a conventional source code debugger 2. Resource information obtaining module 11 can obtain information concerning resources such as a program counter (PC) 25, a general register and a memory, which are accessible by a normal software program, getting from a conventional ISS 20. Meanwhile, resource information displaying module 12 can display the information on a display device 13.

[0008] As a displaying method therein, it is a general practice to display the source codes of a program affixed with marks in positions indicated according to values in the PC so that the ISS indicates which part of the program is in execution as shown in a display example of FIG. 2, or to display a list of contents of the general register as shown in FIG. 3. Moreover, an ISS controlling module 14 can halt execution of the ISS when the process reaches a preset breakpoint. Further, the ISS controlling module 14 can also allow the ISS to execute the program starting from that point in steps (which refer to one instruction of the machine instructions or one line in a program written in a high-level language such as the C language), or alternatively, to execute the program continuously until the next breakpoint. The

program can be verified by confirming how the contents of the register and the memory are changed, by use of the ISS controlling module 14 and the display on the display device 13.

[0009] However, with the spread of the hardware-software co-simulation, an ISS designed to be cycle-accurate, by taking pipeline processing into consideration to achieve timing accuracy with a logic simulator (such an ISS is hereinafter referred to as a "cycle-accurate ISS"), has recently come into use.

[0010] Although the ISS used in the hardware-software co-simulation environment has changed from the conventional ISS to the cycle-accurate ISS, the source code debuggers currently in use continue to fail to account for the pipeline processing. Accordingly, the conventional source code debugger tends to incur the following problems upon displaying a result of the cycle-accurate ISS, which may cause a user to misunderstand the result.

[0011] FIG. 2 shows the state in a display frame of the conventional source code debugger 2, where the ISS is currently halted immediately before the execution of an instruction "LD \$3,0(\$12)" at the address 0x80020200. FIG. 3 shows register information for the state shown in FIG. 2. Now, consideration will be made based on the assumption that a program is executed in steps (which refers to one instruction sentence because the program is written in a machine language).

[0012] FIG. 4 is an example of a result display of the execution of the program one step forward from the state shown in FIG. 2, using the conventional ISS. Since the conventional ISS is controlled to execute the program by each instruction sentence, the instruction "LD \$3,0(\$12)" is completed when the ISS proceeds with the execution of one step. In other words, as shown in register information of FIG. 5, the content of memory indicated by the register \$12 (0x00000064: not shown) is loaded on the register \$3, whereby the PC value is incremented to 0x80020204.

[0013] FIG. 6 shows an example of a result display of the execution of the program one step forward from the state shown in FIG. 2, using the cycle-accurate ISS. A pipeline is assumed to be composed of five stages of "fetch (F)/decode (D)/execute (E)/memory access (M)/write back (W)". Since the cycle-accurate ISS takes the pipeline processing into consideration, the execution of one step refers to the execution of one pipeline stage. Therefore, in FIG. 6, the F stage of the instruction "LD \$3,0(\$12)" is executed as the cycle-accurate ISS proceeds with execution of one step. Accordingly, the cycle-accurate ISS reads the instruction, whereby the PC value is incremented to 0x80020204 as shown in the register information in FIG. 7. Incidentally, the readout of the content of the memory indicated by the register \$12 is carried out at the M stage, and loading on the register \$3 is carried out at the W stage. Therefore, the value of the register \$3 does not change at this point. Accordingly, the register \$3 stays the value 0x00000000.

[0014] In this way, arrows indicating the point of execution point the same address 0x80020204 in both FIG. 4 and FIG. 6. However, the contents of the register \$3 are different when FIG. 5 and FIG. 7 are compared. In order to judge whether the result is correct or wrong, a user always has to grasp the situations at each respective pipeline stage. How-

ever, the operations of the pipeline are quite complicated. Accordingly, there is a problem that the user is prone to misunderstand the value 0x00000000 as the data being loaded on the register \$3. Occurrence of such a problem is not always limited to the case of the LD instruction, but similar problems may occur in the case of a storing instruction or an operating instruction.

[0015] Meanwhile, a display example in FIG. 8 shows the state where the ISS is halted immediately before execution of an instruction "BNE \$2,\$3,0x5" at the address 0x80020200. The instruction "BNE \$2,\$3,0x5" refers to an instruction to compare the values between the register \$2 and the register \$3 and to branch the program toward an instruction 5 steps ahead if the values do not coincide with each other. Here, the values of the register \$2 and the register \$3 are assumed not to be coincident. Now, consideration will be made based on the assumption that the program is executed similarly in each step.

[0016] FIG. 9 shows an example of a result display of the execution of the program from the state shown in FIG. 8 by use of the conventional ISS 20. As the conventional ISS proceeds with the execution of the program for each step, the conventional ISS computes the address of a branch destination (which is 0x80020214) and sets up the value on the PC because the values of the register \$2 and the register \$3 do not coincide with each other. Accordingly, the value of the PC is set to 0x80020214 and an arrow for indicating the point of execution is pointing at the value 0x80020214.

[0017] FIG. 10 shows an example of a result display of the execution of the program one step forward from the state shown in FIG. 8 using the cycle-accurate ISS. As the cycle-accurate ISS only executes the F stage of the BNE instruction, the value of the PC is incremented to 0x80020204. Accordingly, an arrow indicating the point of execution points at the value 0x80020204. In this case, there is a problem that the user is prone to the misunderstanding that a branching condition was not satisfied. A similar problem will also occur in the case of a jump instruction.

[0018] As described above, as the results of the cycle-accurate ISS are often used with the conventional source code debugger 2 under the hardware-software co-simulation environment, the conventional source code debugger 2 cannot display the information concerning the pipeline because the conventional source code debugger 2 does not correspond to the cycle-accurate ISS. Accordingly, the user is always required to grasp the situation at each respective pipeline stage. Otherwise, the problem arises that the user is prone to misunderstand the result.

#### SUMMARY OF THE INVENTION

[0019] A first aspect of the present invention is to provide a source code debugger connected to a cycle-accurate instruction set simulator, comprising: a) a pipeline information obtaining module configured to obtain address information in execution at respective stages on a pipeline from the cycle-accurate instruction set simulator; b) a pipeline information displaying module configured to edit the address information of the pipeline obtained by the pipeline information obtaining module together with the progress of processing of the stages; c) a resource information obtaining module configured to obtain address information of a program in execution together with instruction codes; and d) a

resource information displaying module configured to edit the address information and the instruction codes obtained by the resource information obtaining module.

[0020] A second aspect of the present invention is to provide a source code debugger connected to a cycle-accurate instruction set simulator, comprising: a) means for obtaining address information in execution at respective stages on a pipeline from a cycle-accurate instruction set simulator; b) means for editing the address information of the pipeline obtained by the pipeline information obtaining means together with the progress of processing of the stages; c) means for obtaining address information of a program in execution together with instruction codes; and d) means for editing the address information and the instruction codes obtained by the resource information obtaining means.

[0021] A third aspect of the present invention is to provide a method for use in a system comprising a source code debugger and a cycle-accurate instruction set simulator connected to the source code debugger, comprising: a) obtaining address information in execution at respective stages on a pipeline from the cycle-accurate instruction set simulator; b) editing the address information of the pipeline together with the progress of processing of the stages; c) obtaining address information of a program in execution together with instruction codes; and d) editing the address information and the instruction codes.

[0022] A fourth aspect of the present invention is to provide a computer program product for controlling a system comprising a source code debugger and a cycle-accurate instruction set simulator connected to the source code debugger, comprising: a) instructions configured to obtain, by pipeline information obtaining, address information in execution at respective stages on a pipeline from the cycle-accurate instruction set simulator; b) instructions configured to edit, by pipeline information displaying, the address information of the pipeline together with progress of processing of the stages; c) instructions configured to obtain, by resource information obtaining means, address information of a program in execution together with instruction codes; and d) instructions configured to edit, by resource information displaying module, the address information and the instruction codes.

#### BRIEF DESCRIPTION OF DRAWINGS

[0023] FIG. 1 is a block diagram of a conventional source code debugger.

[0024] FIG. 2 is a source code display frame of the conventional source code debugger immediately before an LD instruction.

[0025] FIG. 3 is a register information display frame in the state of FIG. 2.

[0026] FIG. 4 is a source code display frame of the conventional source code debugger showing a state of executing the LD instruction from the state shown in FIG. 2 by use of a conventional ISS.

[0027] FIG. 5 is a register information display frame in the state of FIG. 4.

[0028] FIG. 6 is a source code display frame of the conventional source code debugger showing a state of executing the LD instruction from the state shown in FIG. 2 by use of a cycle-accurate ISS.

[0029] FIG. 7 is a register information display frame in the state of FIG. 6.

[0030] FIG. 8 is a source code display frame of the conventional source code debugger immediately before a BNE instruction.

[0031] FIG. 9 is a source code display frame of the conventional source code debugger showing a state of executing the BNE instruction from the state shown in FIG. 8 by use of the conventional ISS.

[0032] FIG. 10 is a source code display frame of the conventional source code debugger showing a state of executing the BNE instruction from the state shown in FIG. 8 by use of the cycle-accurate ISS.

[0033] FIG. 11 is a block diagram of a source code debugger according to a first embodiment of the present invention.

[0034] FIG. 12 is a flowchart describing operations of the source code debugger according to the first embodiment of the present invention.

[0035] FIGS. 13A and 13B are examples of a source code display frame of the source code debugger according to the first embodiment of the present invention.

[0036] FIGS. 14A and 14B are examples of a source code display frame of a source code debugger according to a second embodiment of the present invention.

[0037] FIG. 15 is an example of a source code display frame of a source code debugger according to a third embodiment of the present invention.

[0038] FIGS. 16A and 16B are examples of a source code display frame of a source code debugger according to a fourth embodiment of the present invention.

[0039] FIG. 17 is a block diagram of a source code debugger according to a fifth embodiment of the present invention.

[0040] FIG. 18 is a flowchart describing operations of the source code debugger according to the fifth embodiment of the present invention.

[0041] FIG. 19 is an example of a source code display frame of the source code debugger according to the fifth embodiment of the present invention.

#### DETAILED DESCRIPTION OF THE INVENTION

[0042] Various embodiments of the present invention will be described with reference to the accompanying drawings. It is to be noted that the same or similar reference numerals are applied to the same or similar parts and elements throughout the drawings, and the description of the same or similar parts and elements will be omitted or simplified.

[0043] (First Embodiment)

[0044] FIG. 11 is a block diagram showing a constitution of a source code debugger 1 according to a first embodiment of the present invention. The source code debugger 1 according to the first embodiment includes a central processing unit (CPU) 5, a display device 13, an input device 17, a data storage unit 18, and a program storage unit 19. The display device 13, the input device 17, the data storage unit

18, and the program storage unit 19 are respectively connected to the CPU 5, so that information concerning a pipeline in a cycle-accurate instruction set simulator (ISS) 10 can be displayed and the situation of the execution of respective instructions on the pipeline can be thereby grasped adequately. The CPU 5 includes a resource information obtaining module (means) 11, a resource information displaying module (means) 12, an ISS controlling module 14, a pipeline information displaying module (means) 15 and a pipeline information obtaining module (means) 16. The CPU 5 includes the pipeline information displaying module 15 and the pipeline information obtaining module 16 in addition to the constitution of the CPU 6 of the conventional source code debugger 2 shown in FIG. 1. Further, the cycle-accurate ISS 10 is connected to the source code debugger 1. The display device 13 refers to the screen of a monitor or the like. A liquid crystal display (LCD) device, a light-emitting diode (LED) panel, an electroluminescence (EL) panel and the like can be used as the display device 13. The input device 17 refers to an instrument such as a keyboard or a mouse. When an inputting operation is performed with the input device 17, relevant key information is transmitted to the CPU 5. The data storage unit 18 temporarily stores data in progress of calculation or analysis in the course of processing by the CPU 5. The program storage unit 19 stores a program for obtaining the pipeline information or displaying the obtained information, which is to be executed by the CPU 5.

[0045] Meanwhile, a stage information storage unit 21, a stall information storage unit 22 and a register information storage unit 23 are connected to the cycle-accurate ISS 10. The stage information storage unit 21 stores information as to which instruction is in execution at each stage of the pipeline, in order to take the pipeline processing into consideration. The stall information storage unit 22 stores information as to whether each stage is in an executable condition or a stalled condition. And, the register information storage unit 23 stores information concerning the dependency relations of the registers designated as operands of the respective instructions in progress of processing on the pipeline, in order to judge whether the pipeline will stall or not.

[0046] Here, the "cycle-accurate instruction set simulator" refers to an instruction set simulator which simulates an operation of a processor not by the instruction, but by the stage such as fetch (F)/decode (D)/execute (E)/memory access (M)/write back (W) or the like, while taking the pipeline processing into consideration. In the following, "address information of the instruction in execution at each stage of the pipeline" refers to addresses being executed by the respective stages of fetch (F)/decode (D)/execute (E)/memory access (M)/write back (W) and the like. Moreover, "address information of the program in execution" refers to an address of a source code initiated for execution. In the cycle-accurate instruction set simulator, the address information of the program in execution refers to the address in which the fetch (F) stage is initiated.

[0047] The resource information obtaining module 11 embedded in the CPU 5 of the source code debugger 1 according to the first embodiment obtains the address infor-

mation provided by the program counter 25, instruction codes and the like of the program (or the source code) in execution from the cycle-accurate ISS 10. The resource information displaying module 12 of the CPU 5 executes necessary edition processing so as to display the address information, the instruction codes and the like obtained by the resource information obtaining module 11 on the display device 13. Moreover, the pipeline information obtaining module 16 of the CPU 5 obtains the address information of the instruction in execution at each stage of the pipeline, information of registers in execution and the like from the cycle-accurate ISS 10. The pipeline information displaying module 15 of the CPU 5 executes necessary edition processing so as to display the information obtained by the pipeline information obtaining module 16 on the display device 13. Moreover, the ISS controlling module 14 of the CPU 5 controls operations of the cycle-accurate ISS 10.

[0048] According to the source code debugger of the first embodiment, it is possible to display information concerning the pipeline owned by the cycle-accurate ISS in progress of executing software simulation in the hardware-software co-simulation environment. Accordingly, it is possible to grasp situations of the execution of the respective instructions on the pipeline accurately.

[0049] Now, description will be made regarding operations of the source code debugger 1 with reference to FIG. 12.

[0050] (a) In Step S11, the ISS controlling module 14 halts execution by the cycle-accurate ISS 10.

[0051] (b) Next, in Step S12, the resource information obtaining module 11 obtains the address information provided by the PC 25, the instruction codes and the like of the program in execution from the cycle-accurate ISS 10. Then, in Step S13, the resource information displaying module 12 performs edition processing to display on the display device 13, information such as the edition of the address and the instruction codes, addition of an arrow on the source code of the program in a position corresponding to a value owned by the PC 25, or the like.

[0052] (c) Meanwhile, in Step S14, the pipeline information obtaining module 16 obtains the address information of the instruction in execution at each stage of the pipeline. Next, in Step S15, the pipeline information displaying module 15 adds marks relevant to the stages where the respective instructions are executed to positions corresponding to the respective addresses on the source code, and performs editing for displaying on the display device 13. Moreover, the pipeline information displaying module 15 also displays the register information in progress simultaneously.

[0053] (d) Then, in Step S17, the progress of processing the address information and the stage information, the instruction codes and the like are displayed on the display device 13 based on the results of the edition processing in Steps S13 and S15. In this way, it is possible to grasp the current position of the program processing, the progress of processing at the respective stages and the like.

[0054] Now, description will be made regarding a concrete example of a display on the display device 13 with reference to FIG. 13A and FIG. 13B.

[0055] FIG. 13A shows one example of a screen display in the case of displaying the source code on the display device 13. An arrow in front of an address indicates a current position of execution of the program, and signs F, D, E, M and W between the addresses and the instruction codes indicate that the relevant instructions are at the F stage, the D stage, the E stage, the M stage and the W stage, respectively. According to the display in FIG. 13A, it is possible to grasp accurately which stage each instruction is located at. For example, the instruction "LUI \$12,0x8002" at the address 0x800201e4 and the instruction "ORI \$12,0x0400" at the address 0x800201e8 are instructions for initializing the register \$12, which have been executed already. The instruction "LD \$1,0(\$12)" at the address 0x800201ec is the instruction for loading content of the memory designated by the value in the register \$12 onto the register \$1, in which the W stage is about to be executed. The instruction "LD \$2,4(\$12)" at the address 0x800201f0 is the instruction for loading a content of the memory designated by the value in the register \$12 plus 4 onto the register \$2, in which the M stage is about to be executed. The instruction "AND \$2,\$1,\$2" at the address 0x800201f4 is the instruction for operating a logical product (AND) between the value in the register \$1 and the value in the register \$2 and storing the result in the register \$2, in which the E stage is about to be executed. The instruction "LD \$3,8(\$12)" at the address 0x800201f8 is the instruction for loading content of the memory designated by the value in the register \$12 plus 8 onto the register \$3, in which the D stage is about to be executed. The instruction "BNE \$2,\$3,0x5" at the address 0x800201fc is the instruction for comparing the values between the register \$2 and the register \$3 and branching the process toward the instruction five steps ahead if the values do not equal to each other, in which the F stage is about to be executed. The instruction "NOP" at the address 0x80020200 is the instruction for no operation and the instruction "ADD \$3,\$3,\$4" at the address 0x80020204 is the instruction for adding the values between the register \$3 and the register \$4 and storing the result in the register \$3, which are yet to be executed.

[0056] As described above, since the instruction "LD \$3,8(\$12)" at the address 0x800201f8 is on a phase where the D stage is about to be executed, it is easily confirmable that loading onto the register \$3 is not yet completed. Therefore, if the value in the register \$3 is checked in this state, it is possible to avoid the misinterpretation that an incorrect value has been loaded thereon.

[0057] Meanwhile, FIG. 13B shows one example of another mode of displaying the stage information of the pipeline. Whereas the signs F, D, E, M and W are indicated as relevant to the names of the pipeline stages by the pipeline information displaying module in FIG. 13A, the respective stages are displayed in different colors in FIG. 13B. For example, the instruction "BNE \$2,\$3,0x5" at the address 0x800201fc at the F stage is indicated in red, the instruction "LD \$3,8(\$12)" at the address 0x800201f8 at the D stage is indicated in yellow, the instruction "AND \$2,\$1,\$2" at the address 0x800201f4 at the E stage is indicated in blue, the instruction "LD \$2,4(\$12)" at the address 0x800201f0 at the M stage is indicated in green, and the instruction "LD

\$1,0(\$12)” at the address 0x800201ec at the W stage is indicated in orange. Alternatively, the respective stages of the pipeline may be displayed by different hatching patterns.

[0058] According to the source code debugger of the first embodiment, not only the position of execution on the source code but also the stage situation of the pipeline are confirmable. In this way, it is possible to grasp accurate operational conditions without misunderstanding the results.

[0059] (Second Embodiment)

[0060] The source code debugger according to the first embodiment displays only the stage information of the pipeline. On the contrary, a source code debugger according to a second embodiment also displays stall information of the pipeline stage in addition thereto.

[0061] FIG. 14A is a view showing the stall information of the pipeline stages in an identical state and in addition to the display in FIG. 13A.

[0062] FIG. 14A illustrates the example of obtaining the pipeline stage information and the stall information of the respective stages by the pipeline information obtaining module 16 of the CPU 5 in FIG. 11 and displaying the information on the source code by the pipeline information displaying module 15. Upon displaying as to which stages the respective instructions in execution with the pipeline are located at, the stall states of the respective stages are displayed simultaneously. The marks representing the stages affixed with small “s” letters indicate that the relevant stages are stalled. According to FIG. 14A, it is confirmable that the instructions at the address 0x800201ec is at the W stage, the instruction at the address 0x800201f0 is at the M stage, the instruction at the address 0x800201f4 is at the E stage, the instruction at the address 0x800201f8 is at the D stage, and the instruction at the address 0x800201fc is at the F stage. In addition, it is also confirmable that the instructions at the E, D and F stages are stalled.

[0063] For example, FIG. 13A which does not display the stall information indicates that the instruction “AND \$2,\$1, \$2” at the address 0x800201f4 will proceed with the E stage upon execution of the next step. However, one step forward from that state, the E stage will not be actually executed. This is attributable to the fact that the value in the register \$2 designated as the operand is set by the immediately precedent instruction “LD \$2,4(\$12)”; nevertheless, the \$2 value is not fixed yet at this stage because the instruction is in progress of execution of the M stage. Accordingly, execution of the E stage is stalled. As a result, the AND operation between \$1 and \$2 is not executed and \$2 is not updated in the display of FIG. 13A. Such an aspect might incur misunderstanding that the result is incorrect. On the contrary, since the display in FIG. 14A indicates the stalled information of the stages, it is thereby possible to confirm which stages are stalled. Therefore, it is possible to avoid the misunderstanding as described above.

[0064] Moreover, FIG. 14B is an example of indicating stalled stages according to another displaying mode. When the instructions located at the E stage, the D stage and the W stage are stalled, marks indicating the stalled stages are highlighted in reverse video or in a different color.

[0065] According to the source code debugger of the second embodiment, the stage information of the pipeline

and the stall information of the respective stages can be displayed simultaneously, whereby it is possible to grasp a prospective situation upon executing the next step accurately.

[0066] (Third Embodiment)

[0067] A source code debugger according to a third embodiment can be adapted to a processor having a plurality of pipelines for processing instructions.

[0068] FIG. 15 is a view showing one example of source code display of a source code debugger 5 which corresponds to a cycle-accurate ISS 10 of a processor having 2 pipelines for processing instructions. Two pieces of stage information currently in execution can be displayed between addresses and instruction codes. In this case, the cycle-accurate ISS 10 obtains information regarding the instructions in execution at respective stages on a pipeline 1 and a pipeline 2 severally by pipeline information obtaining module 16. Accordingly, it is possible to obtain and display the information regarding the respective pipelines without the addition of any special means to the constitution of the source code debugger 1 shown in FIG. 11.

[0069] In FIG. 15, information regarding the pipeline 1 is displayed on the left side and information regarding the pipeline 2 is displayed on the right side. It is confirmable that the instruction at the address 0x80020200 at the F stage, the instruction at the address 0x800201fc at the D stage, the instruction at the address 0x800201f4 at the E stage and the instruction at the address 0x800201f0 at the M stage regarding the pipeline 1 are in execution, and that the instruction at the address 0x800201f8 at the E stage and the instruction at the address 0x800201ec at the W stage regarding the pipeline 2 are in execution (in which the E stage of the pipeline 1 is stalled).

[0070] The number of the pipelines is not limited to two lines. When the number of the pipelines is three or more lines, it is possible to grasp the situation of all the pipelines simultaneously by adding columns for displaying the additional information to the display in FIG. 15.

[0071] According to the source code debugger of the third embodiment, it is possible to grasp information regarding a plurality of pipelines simultaneously.

[0072] (Fourth Embodiment)

[0073] A source code debugger according to a fourth embodiment can display situations of registers concerning pipeline processing on a screen.

[0074] FIG. 16A shows one example of displaying the register information in the state identical to the state in FIG. 13A. In the event of displaying the source code as shown in FIG. 13A, the cycle-accurate ISS 10 stores information concerning the registers targeted by the instructions in progress of the pipeline processing in order to determine as to whether the pipeline is stalled or not. FIG. 16A shows the example of obtaining the information by pipeline information obtaining module 16 and displaying the register information as a list using pipeline information displaying module 17.

[0075] The reverse video in the drawing indicates that the relevant register is updated in the course of the step immediately preceding. As is clear from FIG. 13A, the instruction



"ORI \$12,0x0400" is completed in the step immediately preceding, and the operation result is stored in the register \$12. Accordingly, the register \$12 is highlighted in reverse video in FIG. 16A. Moreover, hatched display in the drawing indicates the registers currently targeted by the instructions in progress on the pipeline. According to FIG. 13A, the registers currently targeted by the instructions in progress on the pipeline are three registers of \$1 (targeted by the instruction at the address 0x800201ec), \$2 (targeted by the instructions at the addresses 0x800201f0 and 0x800201f4) and \$3 (targeted by the instruction at the address 0x800201f8). Therefore, three registers \$1, \$2 and \$3 are subjected to hatched display in FIG. 16A.

[0076] Meanwhile, FIG. 16B also displays stage situations of the instructions targeting the relevant registers in addition to the registers in hatched display in FIG. 16A. It is evident from FIG. 16B that the instruction at the address 0x800201ec targeting the register \$1 is at the W stage, the instruction at the address 0x800201f0 targeting the register \$2 is at the M stage, and the instruction at the address 0x800201f8 targeting the register \$3 is at the D stage. Here, the register (\$2) targeted by a plurality of instructions is indicated only at the stage (the M stage) for updating the value earliest. However, it is also possible that the relevant register is indicated at all the stages (the M stage and the E stage).

[0077] According to the source code debugger of the fourth embodiment, it is possible to clearly grasp the registers targeted by the instructions currently in progress on the pipeline. Therefore, it is easily possible to judge whether a value displayed on a register information screen is an updated value or a value yet to be updated.

[0078] (Fifth Embodiment)

[0079] A source code debugger according to a fifth embodiment can display a source code display frame and a register information display frame by dividing a frame into pluralities.

[0080] FIG. 17 is a block diagram showing a constitution of a source code debugger 1 according to the fifth embodiment of the present invention. The source code debugger 1 according to the fifth embodiment includes a central processing unit (CPU) 5, a display device 13, an input device 17, a data storage unit 18, and a program storage unit 19, which are severally connected to the CPU 5. The CPU 5 includes resource information obtaining module 11, resource information displaying module 12, ISS controlling module 14, pipeline information displaying module 15, pipeline information obtaining module 16, and screen editing module 26. As compared to the source code debugger 1 according to the first embodiment as shown in FIG. 11, the CPU 5 of the fifth embodiment includes the screen editing module 26 in addition to the constitution of the CPU 5 of the source code debugger 1 according to the first embodiment.

[0081] Since description has been made regarding the resource information obtaining module 11, the resource information displaying module 12, the ISS controlling module 14, the pipeline information displaying module 15 and the pipeline information obtaining module 16 already in the first embodiment, similar explanation will be omitted herein. The screen editing module 26 divides a display frame to be displayed on the display device 13 into pluralities.

[0082] Next, description will be made regarding operations of the source code debugger 1 of the fifth embodiment with reference to FIG. 18.

[0083] (a) Since Steps S11 to S15 are similar to Steps S11 to S15 of FIG. 12 described in the first embodiment, explanation thereof will be omitted herein.

[0084] (b) In Step S16, the screen editing module 26 divides a frame into pluralities for displaying information obtained by the resource information displaying module and the pipeline information displaying module, and the screen editing module 26 performs suitable editing for display on the display device 13.

[0085] (c) Next, in Step S17, the display frame edited by the screen editing module 26 in Step S16 is displayed on the display device 13. In this way, it is possible to grasp the current position of the execution of the program, the progress of processing at the respective stages, register information and the like on one screen.

[0086] FIG. 19 is a view showing one example of displaying a source code frame by dividing into two sections. If the position of the execution of the program is changed by a branching instruction or the like, there is a case depending on a branched address that all the stage information cannot be displayed on the source code at once.

[0087] Here, it is assumed that the frame in FIG. 19 can display a maximum of fifteen lines at once, for example. As the program is branched toward the address 0x800202fc by the branching instruction at the address 0x800201fc, five instructions are in execution on the pipeline at the addresses 0x800201fc, 0x80020200, 0x800202fc, 0x80020300 and 0x80020304. However, since the region from the address 0x800201fc to the address 0x80020304 consists of 67 lines, it is impossible to display all the lines at once. In this case, the source code frame is divided upon display by the screen editing module 26 so that all the related lines can be displayed at once.

[0088] Upon frame division, continuous examples of the addresses for the instruction in execution on the pipeline are checked first and then the frame is divided accordingly to effectuate display, for instance. In FIG. 16A, three stages of F, D and E, and two stages of M and W are severally continuous addresses. Therefore, fourteen lines obtained by subtracting one line for a border from the maximum fifteen lines are divided at a proportion of 2:3, whereby five lines are provided for a first frame (a frame above the border) and nine lines are provided for a second frame (a frame under the border). However, it is to be noted that the mode of frame division is not particularly limited to the above-described mode in the present invention. It is also possible to adopt another mode based on a different algorithm. In addition, the number of display frames is not limited to two divisions, and multiple divisions into more than two frames is also applicable.

[0089] Moreover, the frame subject to division is not limited to the display frame for the source code. It is also possible to divide the register information display frame as described in the fourth embodiment.

[0090] According to the source code debugger of the fifth embodiment, it is possible to display all the instructions in execution on the pipeline on one frame. In this way, it is possible to grasp the situation of the pipeline of the processor completely.

**[0091]** (Other Embodiments)

**[0092]** Although the present invention has been described with reference to certain embodiments, it should be understood that the present invention is not limited to the explanations and the drawings which constitute part of the disclosure. It is obvious to those skilled in the art that various substitutions, examples and operational techniques become feasible based on the teachings of the disclosure.

**[0093]** For example, in the embodiments of the present invention, the positions for indicating the stage information is provided as being between the addresses and the instruction codes and the arrow for indicating the position of the PC is provided as being at the left end. However, those positions are not limited to the above-described modes, and those indication signs can be displayed in any position on the source code as long as the correspondence between the stage information and the instruction codes in execution at those stages is apparent.

**[0094]** Moreover, although the display frame consists of one window according to the embodiments of the present invention, it is also possible to provide for a plurality of windows. In this case, a variety of display combinations becomes feasible, such as a combination of displaying the stage information on one window while displaying the stall information of the stages on the other window, and so forth.

**[0095]** Various modifications will become possible for those skilled in the art after receiving the teachings of the present disclosure without departing from the scope thereof.

What is claimed is:

1. A source code debugger connected to a cycle-accurate instruction set simulator, comprising:

- a pipeline information obtaining module configured to obtain address information in execution at respective stages on a pipeline from the cycle-accurate instruction set simulator;
- a pipeline information displaying module configured to edit the address information of the pipeline obtained by the pipeline information obtaining module together with the progress of processing of the stages;
- a resource information obtaining module configured to obtain address information of a program in execution together with instruction codes; and
- a resource information displaying module configured to edit the address information and the instruction codes obtained by the resource information obtaining module.

2. The source code debugger according to claim 1, wherein the pipeline information obtaining module obtains the address information in execution at the respective stages on a plurality of pipelines from the cycle-accurate instruction set simulator having the plurality of pipelines, and the pipeline information displaying module edits the address information severally regarding the plurality of pipelines together with the progress of processing of the stages.

3. The source code debugger according to claim 1, further comprising a display device configured to display the address information, the progress of processing of the stages and the instruction codes by the pipeline information displaying module and the resource information displaying module in concert.

4. The source code debugger according to claim 1, wherein the pipeline information obtaining module further obtains stalling information of the stages of the respective instructions in execution on the pipeline, and the pipeline information displaying module further edits the stalling information into a source code of the program.

5. The source code debugger according to claim 1, wherein the pipeline information obtaining module further obtains register information targeted by the respective instructions in execution on the pipeline, and the pipeline information displaying module further edits the register information.

6. The source code debugger according to claim 3, further comprising a screen editing module configured to divide a display frame and to display the divided display frame on the display device.

7. A source code debugger connected to a cycle-accurate instruction set simulator, comprising:

means for obtaining address information in execution at respective stages on a pipeline from the cycle-accurate instruction set simulator;

means for editing the address information of the pipeline obtained by the pipeline information obtaining means together with the progress of processing of the stages;

means for obtaining address information of a program in execution together with instruction codes; and

means for editing the address information and the instruction codes obtained by the resource information obtaining means.

8. A method for use in a system comprising a source code debugger and a cycle-accurate instruction set simulator connected to the source code debugger, comprising:

obtaining address information in execution at respective stages on a pipeline from the cycle-accurate instruction set simulator;

editing the address information of the pipeline together with the progress of processing of the stages;

obtaining address information of a program in execution together with instruction codes; and

editing the address information and the instruction codes.

9. The method according to claim 8, further comprising:

obtaining the address information in execution at the respective stages on a plurality of pipelines from the cycle-accurate instruction set simulator having the plurality of pipelines; and

editing the address information in execution and the progress of processing of the stages.

10. The method according to claim 8, further comprising displaying the address information, the progress of processing of the stages and the instruction codes on a display device.

11. The method according to claim 10, further comprising:

obtaining stalling information of the stages of the respective instructions in execution on the pipeline; and

displaying the stalling information on the display device.

**12.** The method according to claim 10, further comprising:

obtaining register information targeted by the respective instructions in execution on the pipeline; and

displaying the register information on the display device.

**13.** The method according to claim 10, further comprising editing so as to display a divided display frame on the display device.

**14.** A computer program product for controlling a system comprising a source code debugger and a cycle-accurate instruction set simulator connected to the source code debugger, comprising:

instructions configured to obtain, by pipeline information obtaining, address information in execution at respective stages on a pipeline from the cycle-accurate instruction set simulator;

instructions configured to edit, by pipeline information displaying, the address information of the pipeline together with progress of processing of the stages;

instructions configured to obtain, by resource information obtaining means, address information of a program in execution together with instruction codes; and

instructions configured to edit, by resource information displaying module, the address information and the instruction codes.

**15.** The computer program product according to claim 14, further comprising:

instructions configured to obtain the address information in execution at the respective stages on a plurality of

pipelines from the cycle-accurate instruction set simulator having the plurality of pipelines; and

instructions configured to edit the address information in execution and the progress of processing of the stages.

**16.** The computer program product according to claim 14, further comprising instructions configured to display the address information, the progress of processing of the stages and the instruction codes on a display device.

**17.** The computer program product according to claim 16, further comprising:

instructions configured to obtain stall information of the stages of the respective instructions in execution on the pipeline; and

instructions configured to display the stall information on the display device.

**18.** The computer program product according to claim 16, further comprising:

instructions configured to obtain register information targeted by the respective instructions in execution on the pipeline; and

instructions configured to display the register information on the display device.

**19.** The computer program product according to claim 16, further comprising instructions configured to edit so as to display a divided display frame on the display device.

\* \* \* \* \*

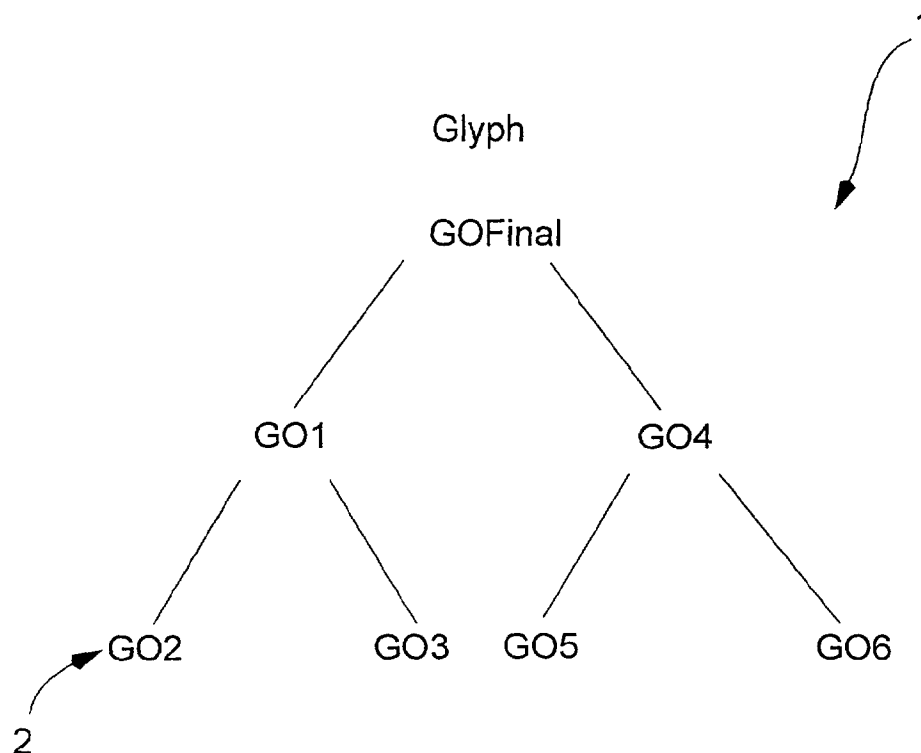
(19) **United States**(12) **Patent Application Publication**  
**HILL et al.**(10) **Pub. No.: US 2002/0130871 A1**(43) **Pub. Date: Sep. 19, 2002**(54) **FONT ARCHITECTURE AND CREATION  
TOOL FOR PRODUCING RICHER TEXT**(52) **U.S. Cl. .... 345/467**(76) Inventors: **GERARD ANTHONY HILL,**  
**CASTLE HILL (AU); CAMERON**  
**BOLITHO BROWNE, BURLEIGH**  
**HEADS (AU); PAUL QUENTIN**  
**SCOTT, PYMBLE (AU); TIMOTHY**  
**MERRICK LONG, LINDFIELD (AU)**(57) **ABSTRACT**

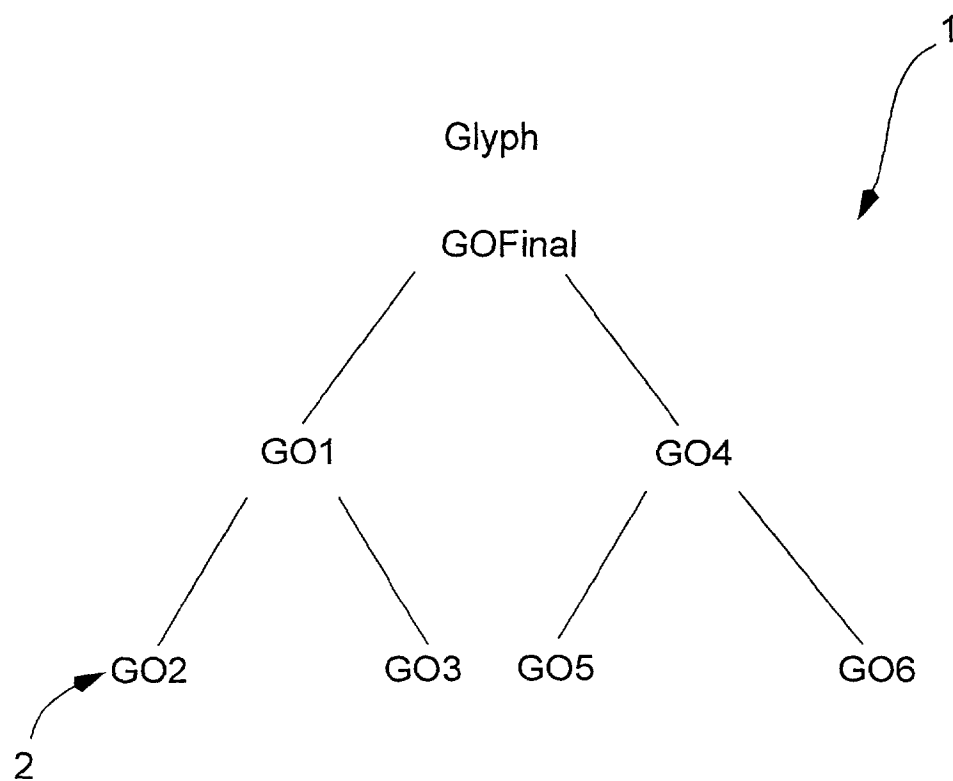
A method of creating a series of font characters (101) on a computer system (150) is disclosed. A series of font outlines (111) and source artwork (112); a series of manipulation tools (116) for the manipulation of aspects of the outlines and artwork. Such an arrangement provides for the creation of substantially arbitrarily complex font structures from the outlines, artwork and manipulation tools. A series of font characters is then formed through the application of the complex font structures to each of a base font outline in the series of font characters. Preferably, the complex font structures can comprise a graphical expression tree of operations (120) to be performed in the creation of a font and the tree includes an outline of a font character. The manipulation tools can include tools for distorting, replacing or compositing the outline of a font and can further include the tools for the application of morphological and non-morphological effects to the font outlines. A data structure for such font creation is also disclosed which includes records (90-97) of attributes of glyphs used to form the outlines including their shape, color, opacity and where appropriate compositing or blending with graphic object or pixel-based images.

Correspondence Address:

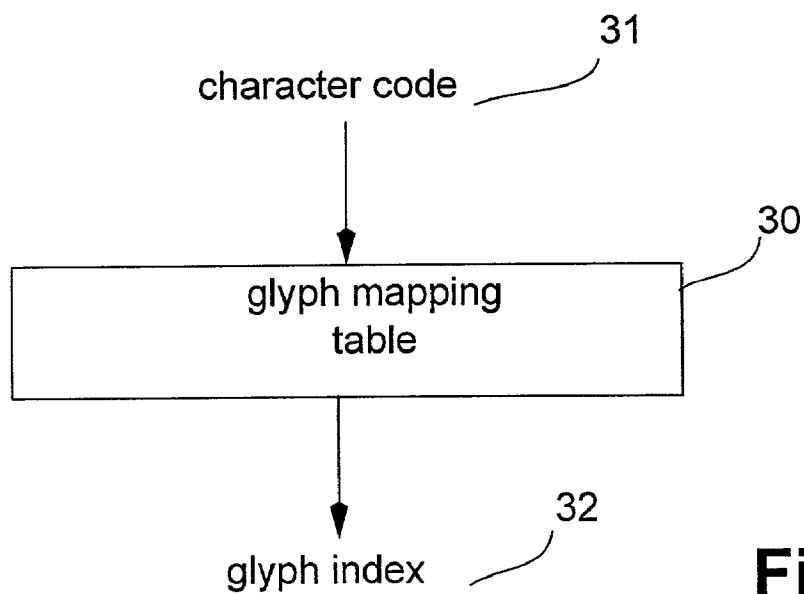
**FITZPATRICK CELLA HARPER & SCINTO**  
**30 ROCKEFELLER PLAZA**  
**NEW YORK, NY 10112 (US)**

( \* ) Notice: This is a publication of a continued prosecution application (CPA) filed under 37 CFR 1.53(d).

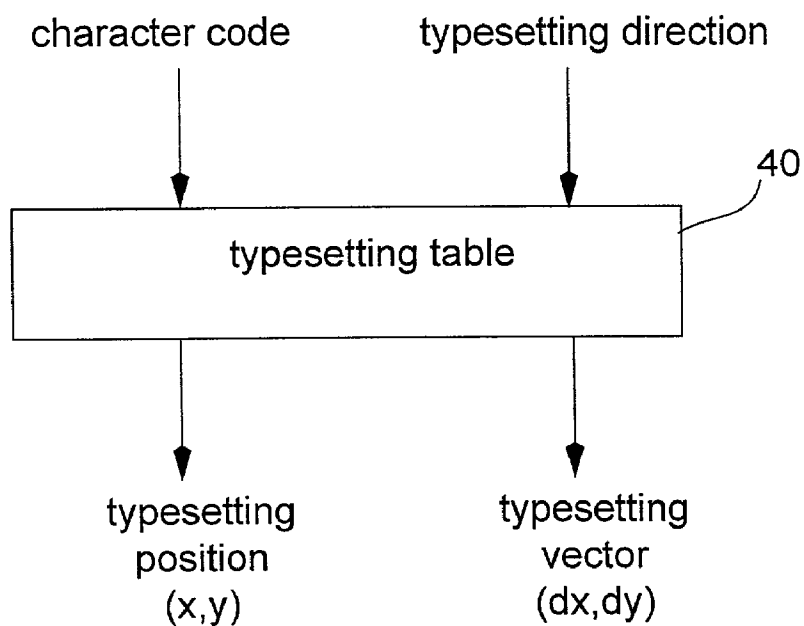
(21) Appl. No.: **09/153,077**(22) Filed: **Sep. 15, 1998**(30) **Foreign Application Priority Data**Sep. 15, 1997 (AU)..... PO9187  
Oct. 1, 1997 (AU)..... PO9566**Publication Classification**(51) **Int. Cl.<sup>7</sup> ..... G06T 11/00**



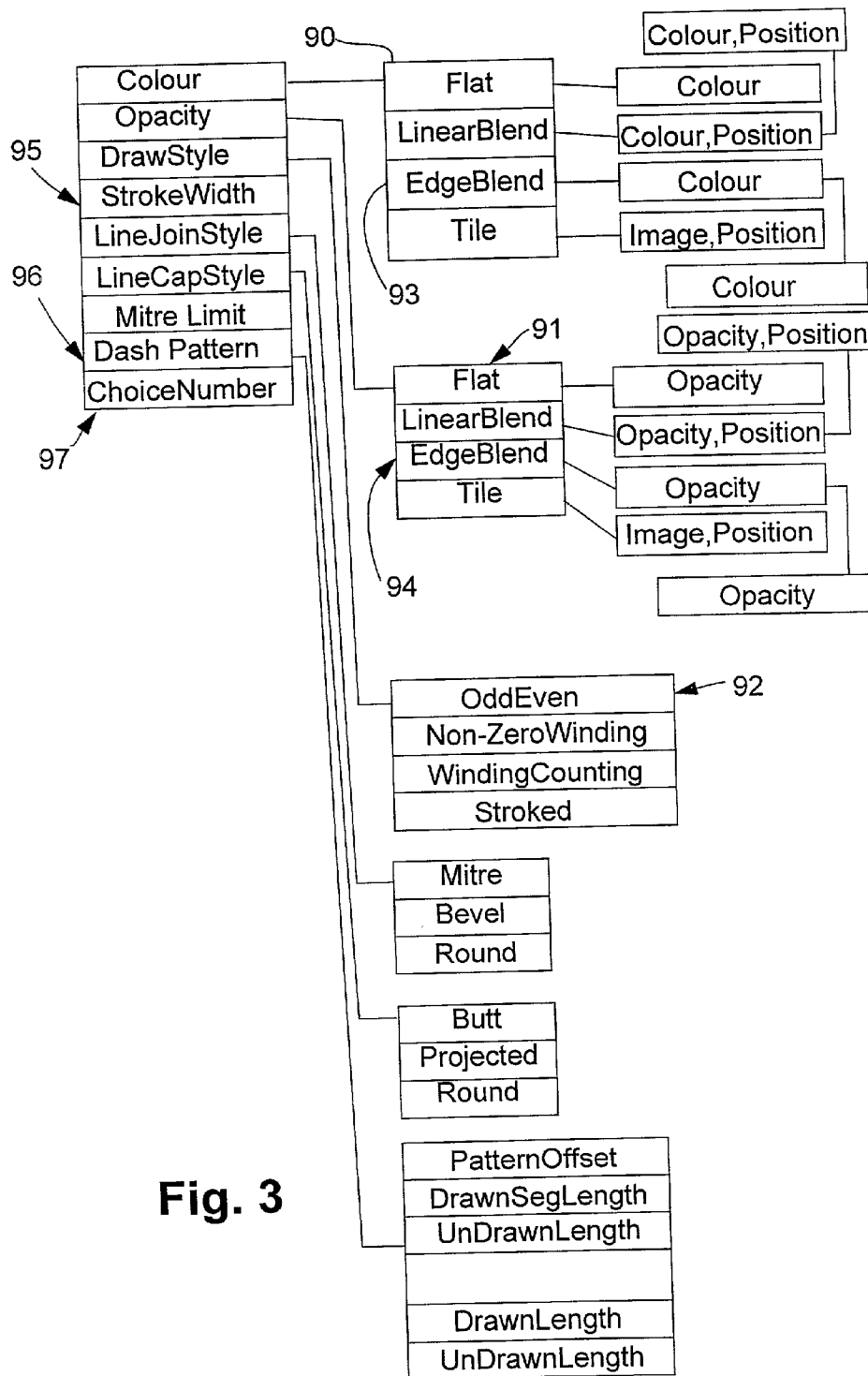
**Fig. 1**



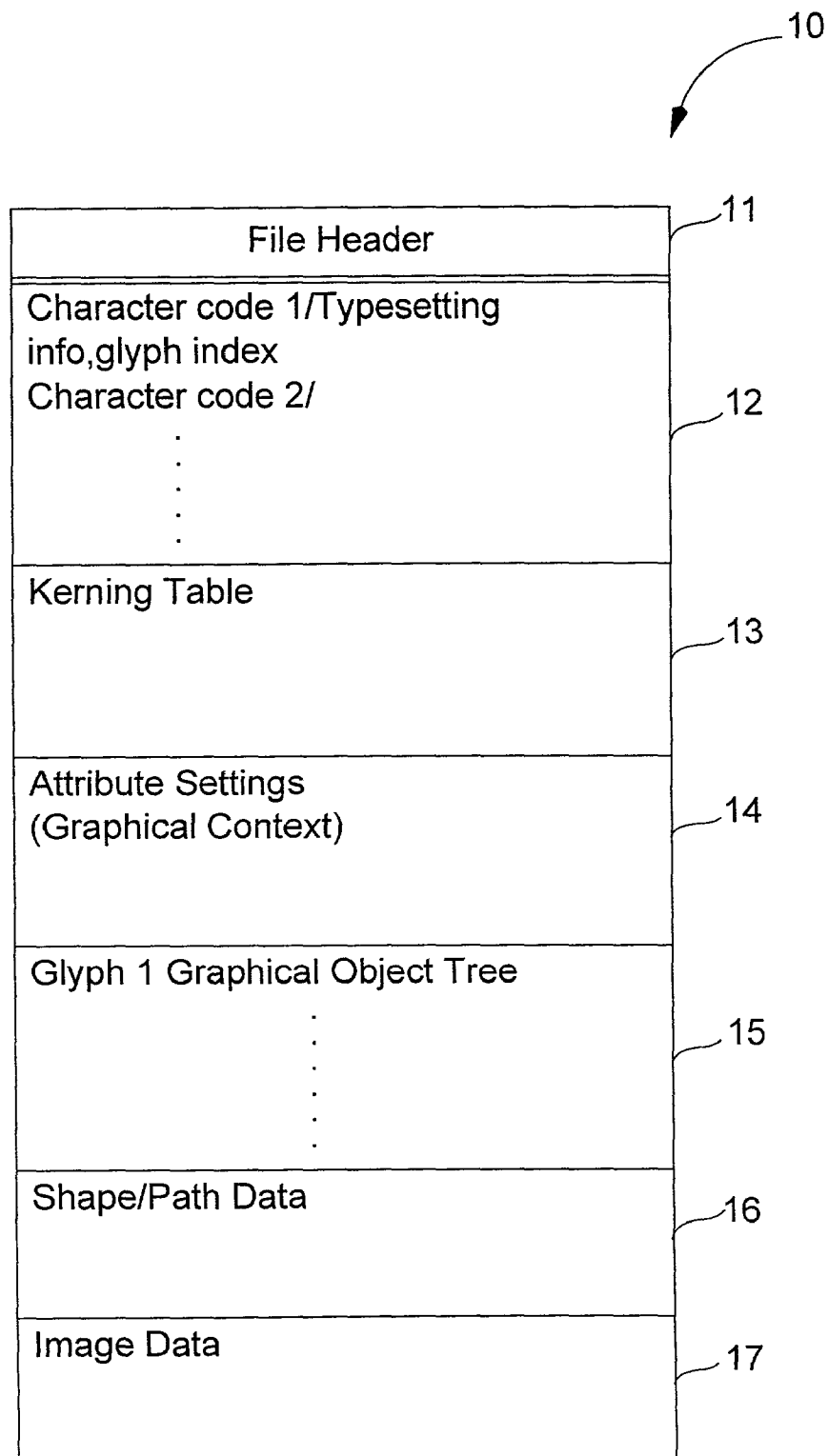
**Fig. 2**



**Fig. 4**

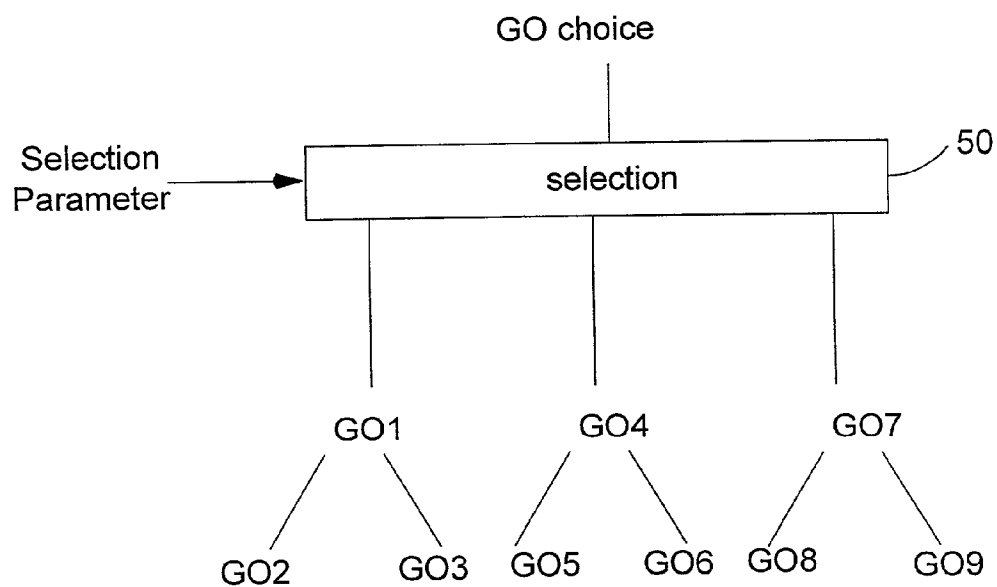


**Fig. 3**

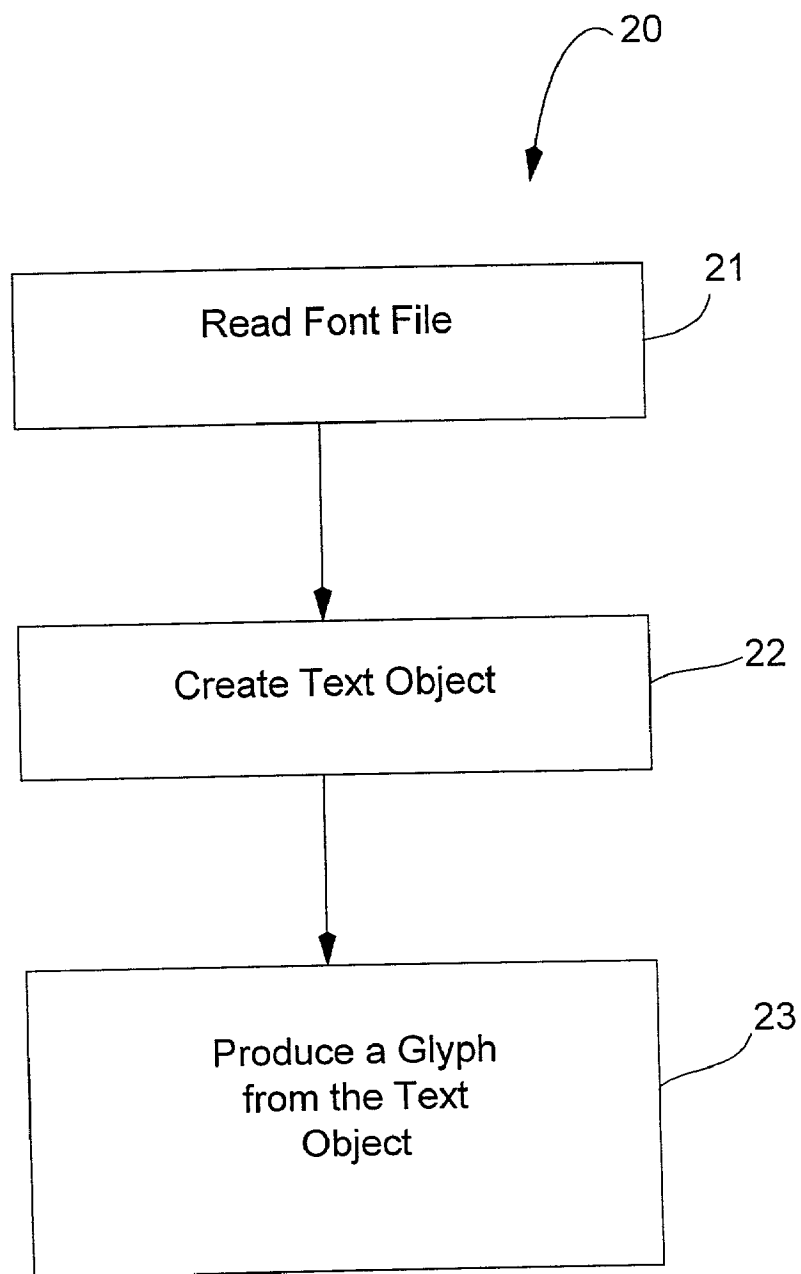


**Fig. 5**

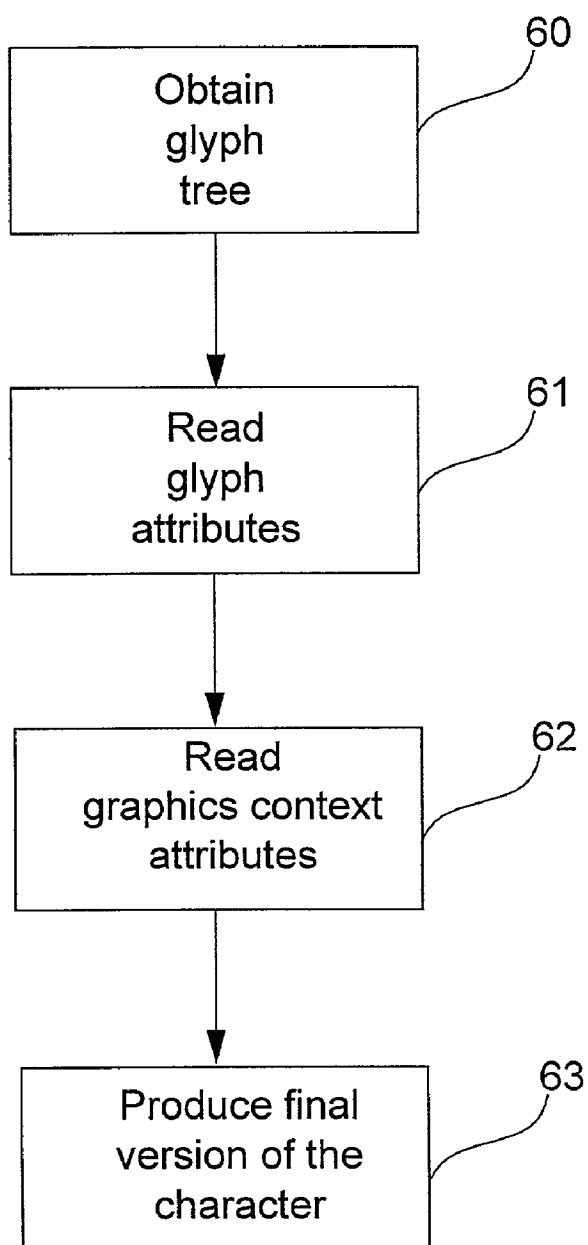




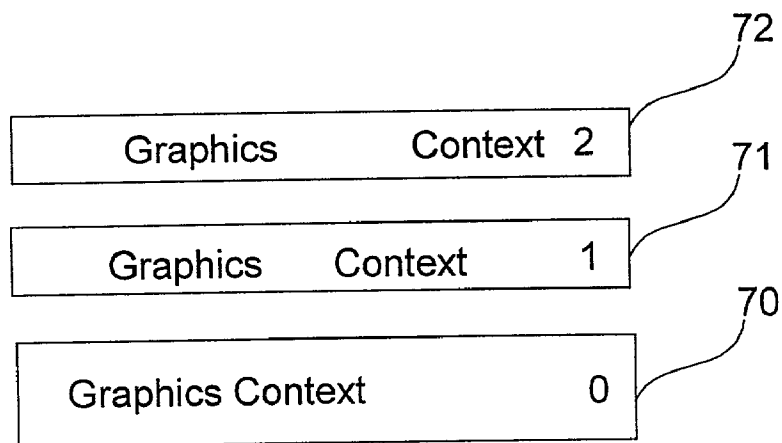
**Fig. 6**



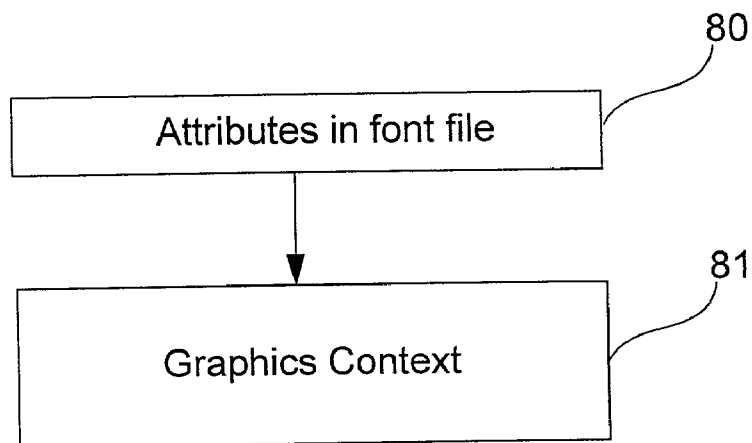
**Fig. 7**



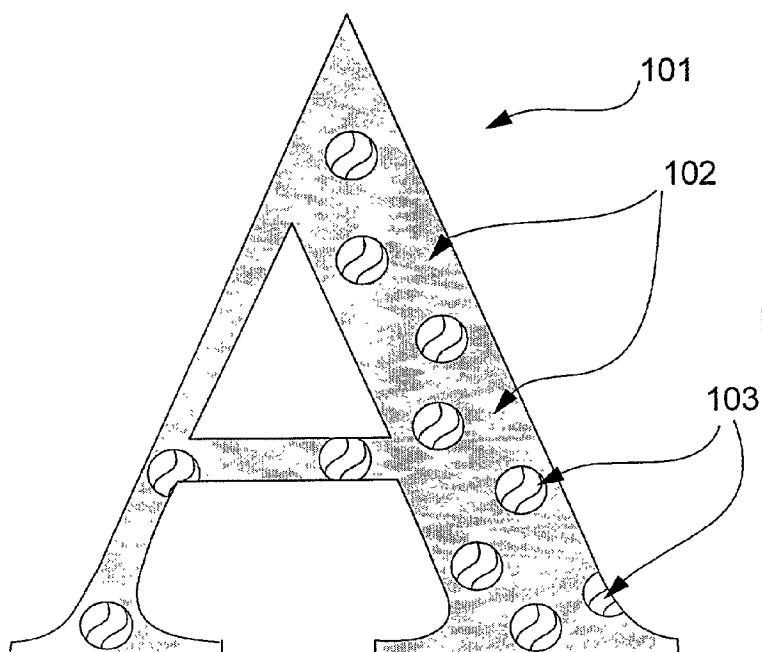
**Fig. 8**



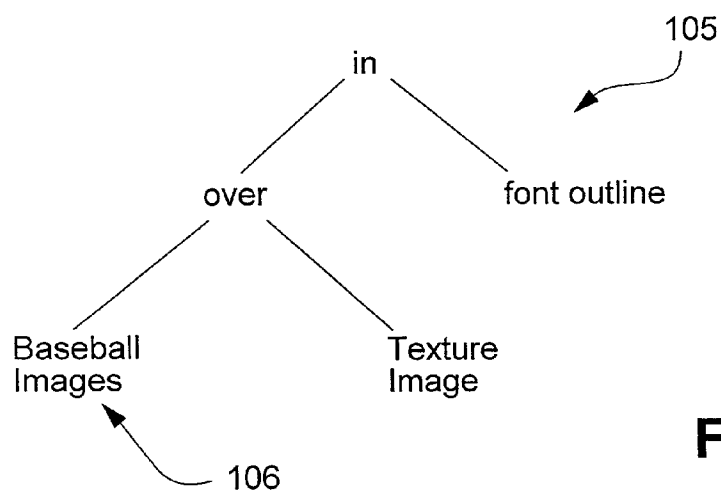
**Fig. 9**



**Fig. 10**



**Fig. 11**



**Fig. 12**

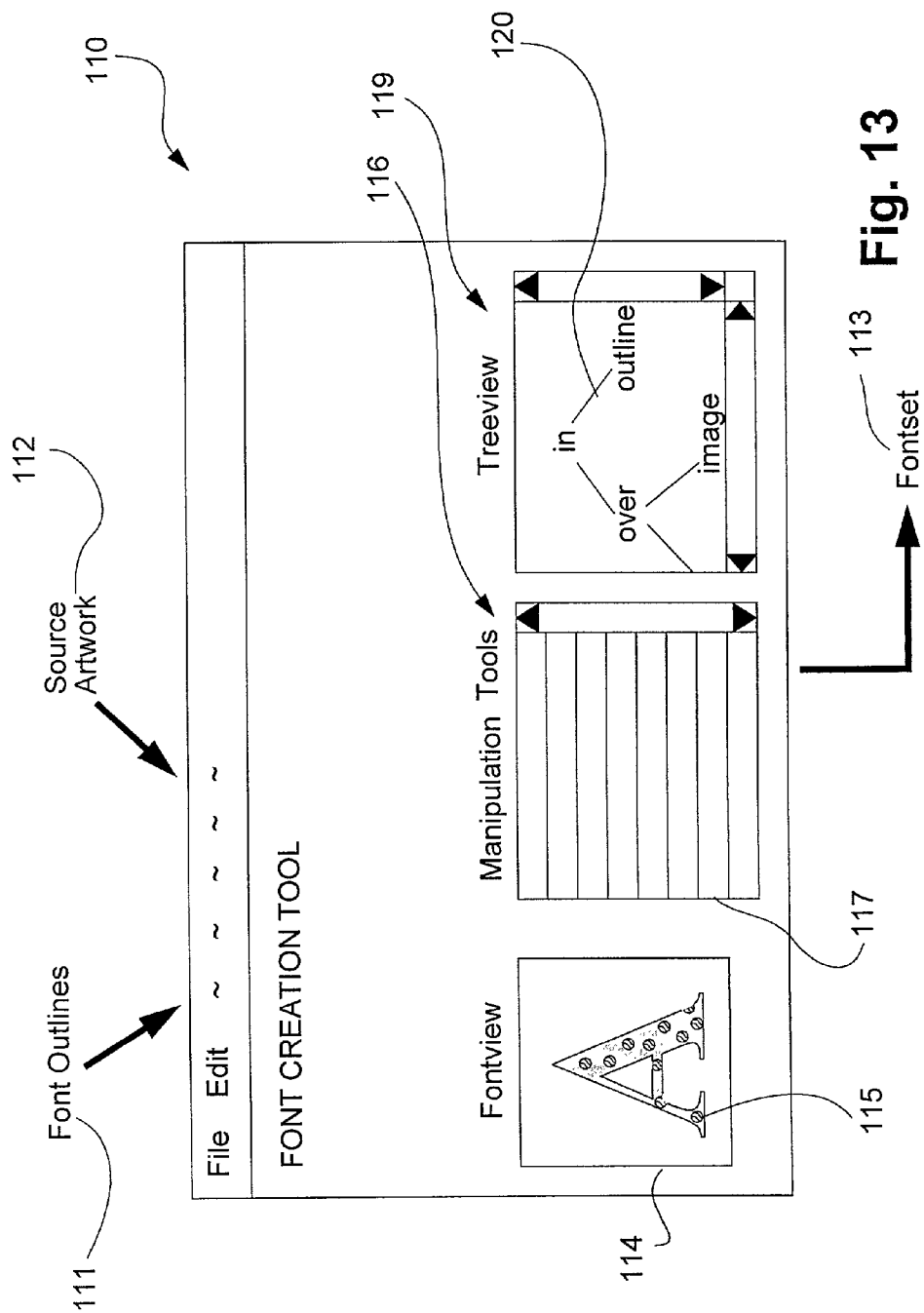


Fig. 13

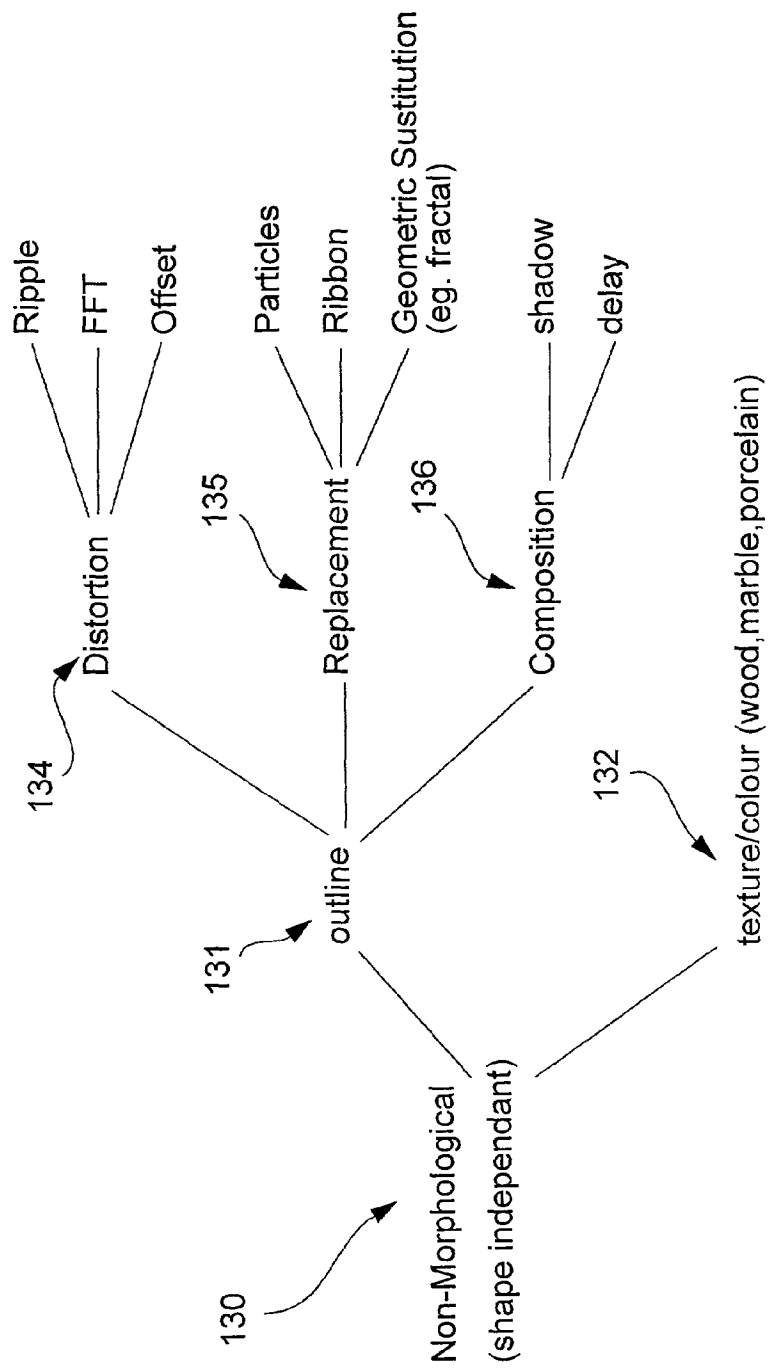
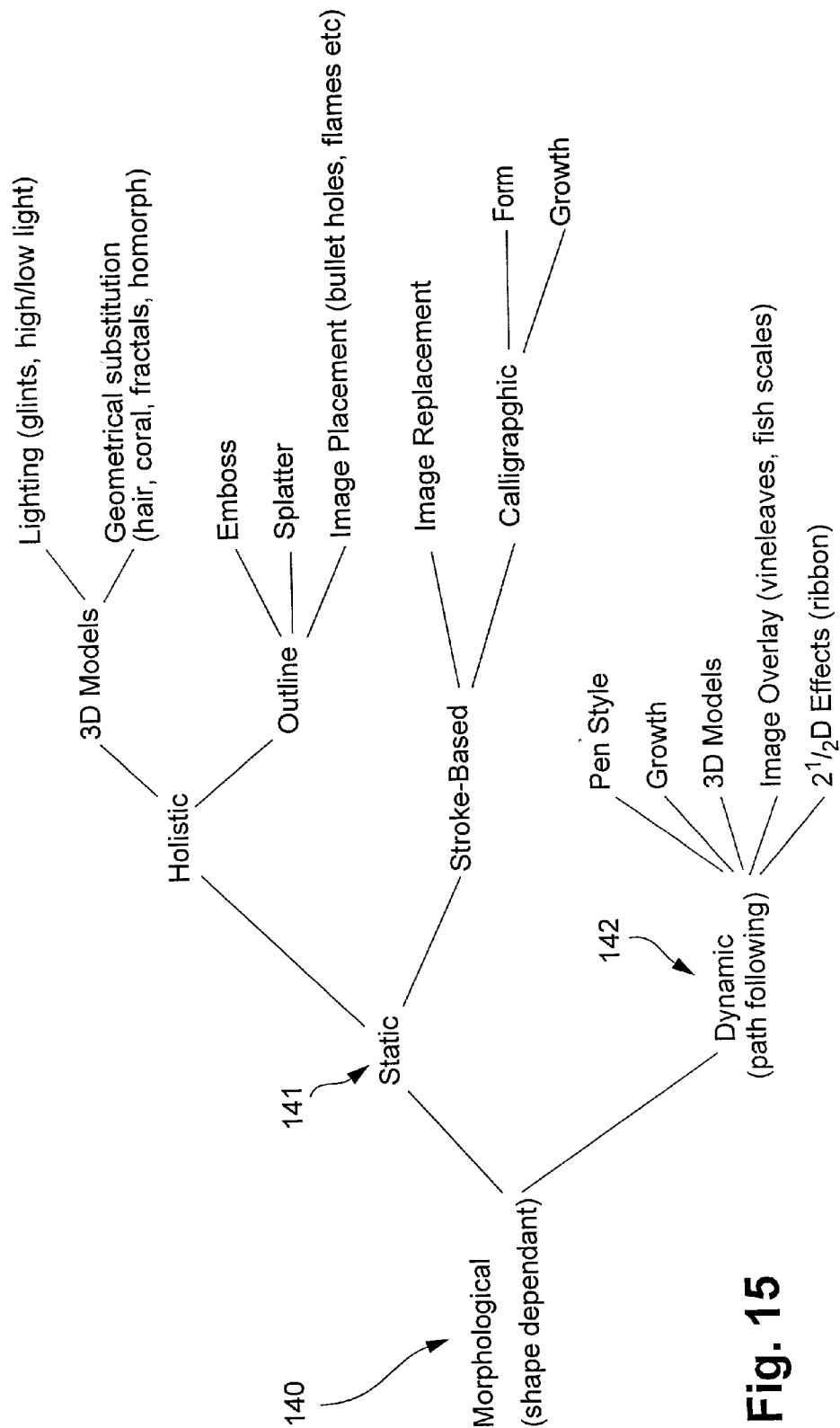
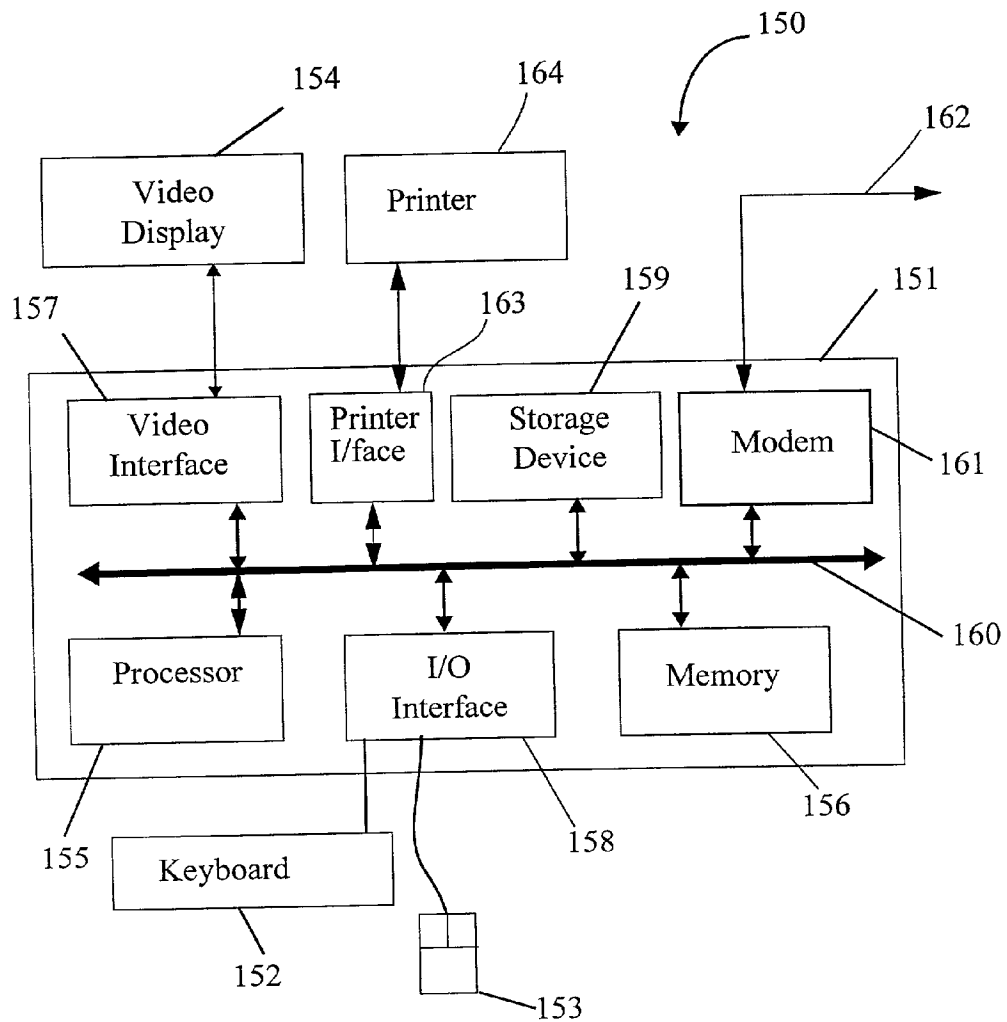


Fig. 14

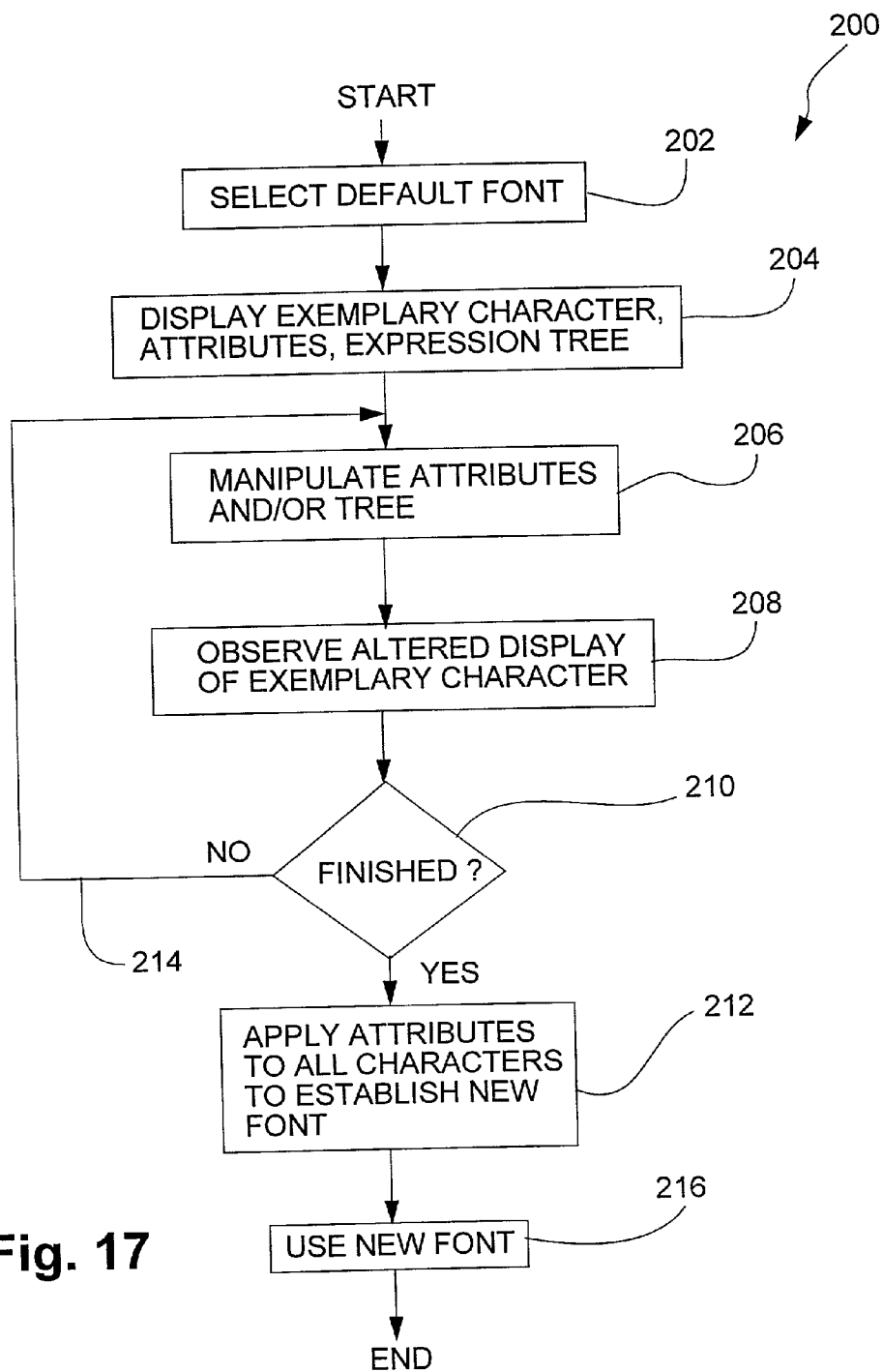


**Fig. 15**





**Fig. 16**



**Fig. 17**

## FONT ARCHITECTURE AND CREATION TOOL FOR PRODUCING RICHER TEXT

### FIELD OF THE INVENTION

[0001] The present invention relates to the field of digital image processing and, in particular, to the creation of images having characters or text in them so as to produce enhanced or superior resulting image fonts. A tool for creating fonts having appealing artistic characteristics is also disclosed.

### BACKGROUND OF THE INVENTION

[0002] Recently, it has become more and more popular to create complex images through the utilisation of a computer system having a high resolution graphics display and a high resolution output printer preferably of a color form. The graphic image production industry is undergoing a rapid development and complex and sophisticated image production tools such as Adobe Photoshop (Trade Mark) are often utilised for the creation of complex images.

[0003] One important aesthetic quality of most images is a character font which conveys text. The design of character fonts is a complex process requiring sophisticated artistic judgements made by the designer. Traditionally, a font has consisted of a bitmap or an outline, the later typically being represented by spline data. The utilisation of font outlines often provides greater flexibility in scaling operations in that the one font can be defined for many different sizes by means of re-scaling of the spline data. Various designed fonts have become extremely popular, for example, Times New Roman, Courier etc.

[0004] Although fonts are well known and utilised in computer image generation programs such as word processing programs, or higher end graphics programs, they are generally lacking in one or more of flexibility, creativity and structure. As the user of the font must work within the pre-defined structure, this often leads to limited or blinkered artistic output results.

[0005] When designing a font, it is necessary to produce designs for each and every character within a font set. This is a laborious and time consuming task, even for the Roman character set, and the languages to which it applies. Further, when designing fonts for other languages, the number of characters within a character set can be extremely large (for example, kanji characters) and hence significant work, labour and expense is involved in the creation of font characters.

[0006] It follows that there is a need to provide a flexible and adaptable font structure which leads to increased levels of flexibility and utilisation. It is also desirable for the font creation process to be substantially automated, whilst still maintaining substantial artistic control over each character, thereby reducing the graphic designer's workload.

### SUMMARY OF THE INVENTION

[0007] It is an object of the present invention to substantially overcome, or at least ameliorate, one or more deficiencies in existing arrangements.

[0008] In accordance with a first aspect of the present invention there is provided a method of incorporating characters from a font into a document or displaying them on a

display medium, said font containing a plurality of glyphs, each glyph comprising one or more shapes, natural or synthesised images, or other glyphs, said method comprising the steps of:

[0009] (a) extracting description of one or more glyphs from the font; and

[0010] (b) rendering the characters onto a display medium or including them as part of a document description.

[0011] In accordance with a second aspect of the present invention there is provided a font structure for use in an image creation system comprising a series of characters wherein each character is made up of a customisable glyph structure, said glyph structure further comprising a series of graphical objects which can be composited together in a predetermined order.

[0012] In accordance with a third aspect of the present invention, there is provided a method of creating a series of font characters on a computer system comprising providing a series of font outlines and source artwork; providing a series of manipulation tools for the manipulation of aspects of the outlines and artwork; providing for the creation of substantially arbitrarily complex font structures from the outlines, artwork and manipulation tools; and creating the series of font characters through the application of the complex font structures to each of a base font outline in the series of font characters.

[0013] Preferably, the complex font structures can comprise a graphical expression tree of operations to be performed in the creation of a font and the tree includes an outline of a font character. The manipulation tools can include tools for distorting, replacing or compositing the outline of a font and can further include the tools for the application of morphological and non-morphological effects to the font outlines. This includes graphical effects that are applied to a set of character outlines, while maintaining the font's readability.

[0014] Preferably, font outlines are:

[0015] (1) derived from existing fonts (eg. True Type);

[0016] (2) generated automatically from letter-form primitives;

[0017] (3) drawn by artists.

### BRIEF DESCRIPTION OF THE DRAWINGS

[0018] Notwithstanding any other forms which may fall within the scope of the present invention, a number of embodiments of the present invention will now be described, by way of example only, with reference to the accompanying drawings in which:

[0019] FIG. 1 illustrates the structure of a character glyph utilising a first embodiment;

[0020] FIG. 2 illustrates the process of deriving a glyph index from a character code;

[0021] FIG. 3 illustrates a layout of the Graphics Context Attributes;

[0022] FIG. 4 illustrates the usage of typesetting information;

[0023] FIG. 5 illustrates the file structure for storage of fonts in accordance with the principles of the first embodiment;

[0024] FIG. 6 illustrate the process of GOB tree selection;

[0025] FIG. 7 illustrates a flow diagram of the steps in accessing a font character in accordance with the first embodiment;

[0026] FIG. 8 illustrates the process of producing a final character;

[0027] FIG. 9 illustrates multiple graphics contexts;

[0028] FIG. 10 illustrates setting attributes of a graphics context;

[0029] FIG. 11 illustrates an example font having interesting characteristic;

[0030] FIG. 12 illustrates an example gob tree structure for the font of FIG. 11;

[0031] FIG. 13 illustrates an example user interface of a second embodiment;

[0032] FIGS. 14 and 15 illustrate example categories for the construction of manipulation tools in accordance with the second embodiment;

[0033] FIG. 16 is a schematic block diagram of a computer system in which the various embodiments of the present invention may be implemented; and

[0034] FIG. 17 is a flowchart of the font creation process according to the preferred embodiment..

## DESCRIPTION OF PREFERRED AND OTHER EMBODIMENTS

### Font Architecture

[0035] In the present embodiment, a font is represented by a series of character "glyphs"; a glyph being a sculptured character or symbol. The glyphs have a complex structure which allows their utilisation in the creation of images having much more complex characteristics.

[0036] Turning now to FIG. 1, there is illustrated schematically the structure of a single glyph 1 of a font according to the present embodiment. In storage, each glyph 1 has a corresponding glyph index number (hereinafter glyph index) utilised to reference the overall glyph. If necessary, there may be one or more character codes or characters which map to a particular glyph index. Hence, the unit of access of characters in the preferred embodiment is by glyph number. The glyph 1 is a description of the visual appearance of a character and is composed of one or more graphical objects 2 which together form a tree structure seen in FIG. 1 describing the appearance of the character corresponding to the glyph index. Such a tree structure describing the appearance of a character is sometimes referred to herein as a graphic object (GOB) tree.

[0037] In FIG. 2, there is illustrated a mapping 30 of a character code 31 to a corresponding glyph index 32 which is carried out utilising a Glyph Mapping Table 30.

[0038] Each graphical object 2 typically includes various information components used in the reproduction of the glyph or font. Examples of such components include Primitives, Attributes of primitives, Operators and Selection.

[0039] Primitives can be formed in a number of ways and are used to characterise an object shape. For example cubic spline paths may be used to define the shape of objects (including character paths). Alternatively, image data in the form of pixel maps (which may or may not be in a compressed format), may define the entire shape.

[0040] Attributes of primitives include various items that provide character and appeal to a primitive.

[0041] FIG. 3 illustrates a layout of Graphic Context Attributes, which attributes can include:

[0042] color and color blend information 90;

[0043] opacity and opacity blend information 91;

[0044] edge blends of the edges of objects eg. 93,94;

[0045] one or more transformation matrices (not illustrated) to be applied to paths and images; and

[0046] draw style 92 and stroking parameters for paths 95,96.

[0047] Binary Operators are used together for the compositing together of two or more graphical objects. These can include OVER, IN, OUT, ATOP, XOR, PLUSC, PLUSW and are discussed in the well known article, "Compositing Digital Images", Porter, T; Duff, T; *Computer Graphics* Vol. 18 No. 3, July 1984, pages 253-259. Other compositing operators include NOT-XOR, PLUS, MINUS, AND, NAND, OR, NOR, NOT, CLEAR, SET, and COPY

[0048] Selection provides a choice of attached graphic objects based on the present environment state.

[0049] It will be evident that any primitive utilised in a graphics object can be greatly parameterised. This means that the corresponding feature can be referred to by altering a parameter in the graphics context and having the resulting value changed when the character is used. The decision to allow objects and features to be parameterised provides great flexibility in the hands of the font designer in the creation of complex fonts. A default value is preferably provided for each feature that is parameterised.

[0050] Returning to FIG. 1, graphics object having a selection choice can form a node (eg. G01) of the GOB tree and such provides a method for selecting an object from a group of descendant graphics objects (eg. G02, G03) provided with the font, depending on a desired graphical context. The selection can be achieved in conjunction with any parameters in the graphics context. An example of the selection process is described below with reference to FIG. 6.

[0051] Parameters in primitives can be used as groups of settings to be combined to achieve a particular effect. In order to take advantage of this, the font as noted above may include a section for attributes. The attribute section contains parameter settings that can be transferred to the graphics context upon a user request.

[0052] Attributes in a glyph that can be affected (see FIG. 3) can include:

- [0053] color;
- [0054] opacity;
- [0055] draw style;
- [0056] stroke width;
- [0057] line join style;
- [0058] line cap style;
- [0059] meter limit;
- [0060] dash pattern; and
- [0061] choice number.

[0062] All color and opacity values can preferably be parameterised across a whole font. This means that only a few colours need to be chosen to make a whole font appear color matched and there is freedom for a font designer to use different parameters for each character if this is required. Each unique color value to be parametered can be taken from a different graphics context number (GC number) as shown in FIG. 9.

[0063] As shown in FIG. 4, each glyph 1 can have typesetting information 40 for its vertical and horizontal typesetting directions taken from a font file. For example, the following can be provided:

- [0064] horizontal typesetting position (x, y);
- [0065] horizontal typesetting vector (dx, dy);
- [0066] vertical typesetting position (x, y); and
- [0067] vertical typesetting vector (dx, dy).

[0068] Through the utilization of the glyph structure 1 of FIG. 1, arbitrarily complex font structures can be created. A font can be produced which contains vastly more information than a traditional font and it therefore allows for much more detail in its appearance. However, traditional font characters can also be readily stored as glyph graphical objects so there is flexibility to make them as simple or as complex as is required.

[0069] The glyph structure can therefore be adapted to be similar in operation to a conventional font so that it can be used interchangeably. The extensions to a graphics language which allow the user to control aspects of the appearance of characters in a glyph structure are purely optional.

[0070] The user is then able to customise the appearance of the resulting glyph characters to obtain a desired appearance. Each font has a default "appearance" and the font designer can include arbitrary customisable parameters to allow the user to alter the default appearance to a custom appearance.

[0071] A font can therefore be created from glyphs 1 and stored in a "font file" 10 as shown in FIG. 5 which may contain the following elements:

- [0072] font header 11;
- [0073] character code, glyph index and typesetting information 12 for each character;
- [0074] Kerning table 13;

[0075] Graphics Context attribute settings 14 for the Graphics Context of the Font;

[0076] Tree structures 15 (or GOB trees) of graphical objects for each glyph including pointers to shape 16 and image data 17;

[0077] Shape data 16; and

[0078] Image data 17.

[0079] The structure of a font from the user's point of view is as a collection of graphic objects 2, of which there are at least one per glyph 1. The graphic objects 2 can contain elements that are used in other graphic objects. Further mechanisms can be provided for altering the appearance of a graphic object by altering the Graphics Context attributes 14. As illustrated in FIG. 6, by changing the choice number and utilising the selection parameter in selection mechanism 50, it is possible to obtain a wide variety of appearances for a single character. Customisation of parameters associated with the glyph can further give the font a unique appearance.

[0080] FIG. 7 shows a process 20 that is implemented to process a font character in accordance with the present embodiment. The process 20 includes the steps of reading the font file 21, creating 22 a text object referring to the correct character, and producing a corresponding glyph 23 from the text object.

[0081] When the font file is read it can be converted to an internal representation, for example a "cached" version, and the cached version of the font can be utilised.

[0082] FIG. 8 illustrates the formation of a graphic object resulting from processing a GOB tree for each character. Graphical objects can contain fully defined instances of characters and are accessed at step 60 where a glyph tree is obtained. In step 61, all of the aspects of the appearance of the character(s) are fixed which permits the glyph attributes to be read. Conversion to a final graphical object at step 63 involves applying in step 62 the read graphics context attributes to all parameters. This stage is where the user can exercise control over the appearance of characters.

[0083] One of the important features of the present font architecture is that the font appearance can be customised by the user of the font. The way this can be achieved is by setting the attributes in the graphics context (FIG. 3). Further, as illustrated in FIG. 9, the graphics context can be extended so that it has multiple instances 70-72, rather than just one. Each instance can be referred to by a graphics context number (GC number). These can have all of the possible attributes available for the font and can be used to customise characters in the font. The attributes at any GC number are initialised by setting them up at the base level of the graphics context (GC number 0) 70 and then copying them to the new GC number. Alternatively, the attributes can be loaded from attributes in the font file as shown in FIG. 10. A graphics context at any GC number can be copied back to the base level. For example, the following commands can be used to support graphics context numbers:

[0084] 1. gc\_copy("attr-name",gcnum) or

[0085] gc\_copy("attr-name",gcnum1,gcnum2).

[0086] This command is used to copy an attribute (attr-name) from the graphics context number zero (gcnum0 (70)

to the number specified. In the second case, the attribute is copied from gnum2 (72) to gnum1 (71).

[0087] 2. gc\_swap("attr-name",gnum) or

[0088] gc\_swap("attr-name",gnum1,gnum2).

[0089] This command is used to swap an attribute (attr-name) from the graphics context number zero with the number specified. In the second case the attribute is swapped between gnum1 and gnum2.

[0090] 3. gc\_load\_font\_defaults( ).

[0091] As illustrated in FIG. 10, the gc\_load\_font\_defaults command is used to set the attributes in the graphics context 81 from the defaults in the font 80. In this case the attributes are typically loaded from the font file.

[0092] 4. gc\_clear( ).

[0093] The gc\_clear command is used to clear all of the graphics contexts except for gnum0.

#### Choice Number

[0094] An attribute described with reference to FIG. 3 and included in the graphics context attributes is the choice number attribute 97. The choice number attribute 97 can be read during the processing of a font character file of FIG. 10. The choice number attribute is preferably a numeric value. The commands that affect choice number can include:

[0095] gc\_choice\_num(option). This command sets the choice number for GC number zero 70 to the (numeric) value specified. If the value specified is zero then this can be considered to be the default font option.

[0096] option=gc\_choice\_num( ). This command returns the value of the choice number for GC number zero 70.

[0097] It can be seen from the foregoing description that a system of font creation is provided which offers substantially greater flexibility than that known in the prior art, and provides for arbitrarily complex font structures. The font structures can be readily adaptable or amendable by the font creator or the font and user in accordance with requirements.

#### Font Creation Tool

[0098] In this embodiment, a tool is provided for the creation of sophisticated fonts which allow for the creation of a structure from which fonts can be derived.

[0099] Although the principles of the instant embodiment have general applicability, and are in particular applicable to "bit map" generated end fonts, the instant embodiment preferably utilises the font architecture system described above. Of course, it is possible to create a font set for other characters utilising other complex image creation packages such as Photoshop, especially when macro languages are utilised within such packages.

[0100] Turning initially to FIG. 11, there is shown an example of a complex font comprising the Times Roman character font "A" 101 which has been manipulated so as to contain within its border a complex image structure comprising a background image 102 on top of which is placed a series of round ball like objects 103.

[0101] FIG. 12, shows a corresponding "graphical object tree" (GOB) 105 that might be created in accordance with the principles of the aforementioned specification corresponding for creating the font character 101. The expression tree 105 includes a number of operators having operands which, together, are composited to create a final font. The compositing process can be performed in real time or rendered off line, depending on requirements. Amendments may be made to any part of the GOB tree 5. For example, it may be desirable to provide for the user to select which ball object 106 is placed within the font character 1 depending on requirements. Hence the ball object 106 could comprise a clip art and could be selected from a list of different ball objects (a baseball for example) depending on the font creation characteristics required.

[0102] Turning now to FIG. 13, there is illustrated one form of user interface for a font creation tool 110 as constructed in accordance with the instant embodiment. The font creation tool 110 can be implemented utilising standard software development tools such as Microsoft Visual C++ Developer's environment. The font creation tool 110 takes, as its input, font outlines which can be in the form of standard true type outlines 111 in addition to source artwork which can comprise images and clip art. The output of the font creation tool is a font set 113 which can correspond to a character set in accordance with requirements. Alternatively, for example, a full Unicode character set output can be provided. The font creation tool 110 is based around a user interface which includes a window 114 for viewing sample end fonts 115. The fonts are manipulated by a series of manipulation tools 116 which can be accessed via a user interface list 117. The accessing of items in the list 117 can result in a pop up window for the setting of manipulation tool variables in accordance with requirements. Additionally, a tree view panel 119 can be provided for the display and manipulation of GOB trees 120. Additional user interfaces (not shown) can be provided for the manipulation of the elements within the tree view 119 for the creation of arbitrary trees so as to construct alternative font arrangements.

[0103] Through the provision of a large range of manipulation tools 116, each tool having a number of independent variables which can be set, in addition to the creation of arbitrarily complex GOB tree structures, a means can be provided for the creation of a large number of font structures with the rapid testing of the independent variables so as to provide for the most suitable end results, in which sample font is displayed in the font view window 114 for immediate appraisal. Of course, the user interface of the preferred embodiment can be readily adaptable and malleable in accordance with changing requirements and added developments.

[0104] Ideally, the font character outline is utilised in, at least, one portion of the tree view 119. Hence, an output font set 113 can be created by means of substitution of the outline pass within the GOB tree 120 for each character so as to produce a corresponding output character.

[0105] The manipulation tools 116 can be many and various. The manipulation tools 116 can be divided into those which are non morphological (shape independent) and those which are morphological (shape dependant). Graphs of the subject groupings under each of these two categories

is illustrated in **FIGS. 14 and 15**. In **FIG. 14**, there is illustrated the non morphological categories **130** which can be further divided into operations applied to the outline **131** of a font in addition to those applied to the texture or color of a font **132**. The outline modifications can include distortions **134** such as applying outline ripples, altering the Fourier components or offsetting the fonts outlines. Replacement **135** can include such things as replacing the font outline with a series of structure such as particles or ribbons or applying a geometric substitution to the outline. Further, composition of the outline **136** can comprise operations such as shadows and delay functions. Texture or color operations **132** can include the selection of a substantial range of textures and colours for a particular font.

[0106] Turning now to **FIG. 15**, there is illustrated an example graph of the categories of morphological operations **140** which can be applied to a font. These can include both static operations **141** and dynamic operations **142**. The static operations **141** are shown divided into holistic and stroke based operations. The holistic operations can include the application of three dimensional models to a font and the application of outline algorithms to a font. The stroked based operations can include image replacement and calligraphic operations. The dynamic operations **142** can be applied to the pen style, the structure of growths on the font, 3D models of a font, image overlays and any  $2\frac{1}{2}$  dimensional effects such as ribbons etc.

[0107] The outline of categories of **FIG. 14** and **FIG. 15** merely represents some of the manipulations that can be applied to a font so as to produce interesting effects. Further, the manipulations can be applied to sub-parts of a font and can be utilised in the construction of the GOB expressions tree representing the font.

[0108] It will therefore be evident to those skilled in the art that through the utilisation of an interface similar to that depicted schematically in **FIG. 13**, extremely complex and interesting effects can be built up and tested on sample characters before the subsequent creation of a complete font set. The utilisation of the GOB tree structure also allows for the creation of arbitrary complex graphical images which can then be suitably utilised in font creations.

[0109] The various embodiments of the present invention may be practiced using a personal computer system **150** such as that shown in **FIG. 16**. The computer system **150** includes a computer module **151**, a video display monitor **154** and one or more input devices such as a mouse pointing device **153** and a keyboard **152**, connected to the computer module **151**. The computer system **150** may be connected to one or more other computers, a computer network such as a LAN, WAN or the Internet, using a communication link **162** and an associated modem device **161**, typically but not necessarily arranged within the computer module **151**. Further, any of several types of hard copy reproduction output devices **164**, including plotters, printers, laser printers, may be connected to the computer module **151** via an appropriate interface **163**.

[0110] The computer module **155** has one or more central processing units (CPU or processor) **155**, a memory module **156** including volatile random access memory (RAM), static RAM or cache and read-only memory (ROM), and an input/output (I/O) interface **158** connected to the input devices **152**, **153**. Storage device(s) **159** provide for non-

volatile storage of data and a video interface/adaptor **157** connects to the video display monitor **154** to provide video signals from the computer module **151** for display on the video display monitor **154**. The storage device(s) **159** may comprise one or more of a floppy disc, a hard disc drive, a magneto-optical disc drive, magnetic tape, CD-ROM and/or any other of a number of non-volatile storage devices. The components **155** to **159** and **161** shown in **FIG. 16** are coupled to each other via a bus **160** typically including data, address, and control buses and interact to operate in a substantially conventional manner corresponding to known systems such as the IBM PC/AT or compatible arrangements, one of the Apple Macintosh (TM) family of computers, Sun Sparcstation (TM), or the like. The overall structure and individual components of the computer system **150** is essentially conventional and would be well known to persons skilled in the art. Thus, the system **150** is simply provided for illustrative purposes and other configurations can be employed without departing from the scope and spirit of the invention.

[0111] The preferred embodiments typically operate as software running on the computer system **150** and incorporate a series of instructions typically resident in the storage device **159** (eg. hard disk) but normally operative from the RAM **156**. The software may alternatively be sourced from the computer network and is operative under user control to vary character fonts interactively using the display monitor **154** and for reproduction purposes on the display monitor **154** or via the printer **164**.

[0112] Font creation be achieved in a number of ways, one of which will now be described with reference to **FIGS. 13, 16** and **17**. In this preferred embodiment an application program running from the hard disk **159** of the computer system **150** implements a method **200** which initially prompts the user to select a default font at step **202**. This may, for example as shown in **FIG. 13** results in the selection of the Times New Roman Font. The system then acts to display an exemplary character and its corresponding attributes and expression tree in the windows **114**, **116** and **119** respectively at step **204**. Having presented the default font, the user may then manipulate the attributes **117** within the window **116** and/or the expression tree **120** to achieve a desired visual effect. Preferably, as manipulation occurs, the altered font is updated in the window **114** permitting the user to observe the amendments, this being step **208**. If the user is not satisfied with the altered font displayed by the exemplary character, the method permits a return via path **214** to step **206** where further manipulation may take place. If satisfied, in step **212** the user acknowledges the new font which is then applied to all characters within the corresponding character set. In the illustrated example, the outline shape off the Times New Roman character set is not altered, but the color fill of the characters within the set is changed to reproduce a textured image over which are composited an arrangement of balls, as described above. The new character set is then made available for general use at step **216**, and this may include any one or more of a plethora of uses of the font, for example to alter the text within a word processing document, or to create a banner over a document displayed in a conventional font. The windows **110**, **114**, **116** and **119** are preferably displayed on the video display **154** of the computer system **150** and the new font may be stored for use on the hard disk **159** for example. Alternatively the new font may be used directly with the printing of the desired

document or made available to other computers via the computer network. The default font need not be a traditional font as such, but may incorporate a data format corresponding to the new architecture described above which thus permits corresponding new fonts to be supplied to the computer system **150**, either via the network or by floppy disk for example. The user may then use those new fonts directly in documents or alternatively amend the new fonts in the manner described above to produce further fonts.

**[0113]** It would be appreciated by a person skilled in the art that numerous variations and/or modifications may be made to the present invention as shown in the specific embodiments without departing from the spirit or scope of the invention as broadly described. The described embodiments are, therefore, to be considered in all respects to be illustrative and not restrictive.

1. A method of incorporating characters from a font into a document or displaying them on a display medium, said font containing a plurality of glyphs, each glyph can include shapes, natural or synthesised images, or other glyphs, said method comprising the steps of:

- (a) extracting description of one or more glyphs from the font; and
- (b) rendering the characters onto a display medium or including them as part of a document description.

2. A method according to claim 1, wherein the components of said glyphs are organised into a graph data structure.

3. A method according to claim 2, wherein the components of said glyphs are organised into a hierarchical structure.

4. A method according to claim 3, wherein the components of said glyphs are organised into a tree structure.

5. A method according to claim 1, wherein characteristics of said glyphs have the property that during rendering their value is set by:

- (a) the font and are as set by the glyph designer; or
- (b) the user of the font who has chosen to override the value set in the font.

6. A method according to claim 1, wherein components of said glyph picture description are combined using compositing operators during instantiation.

7. A method according to claim 6, wherein said compositing operators may operate on at least one component.

8. A method according to claim 7, wherein said compositing operators utilise transparency values associated with said components of said glyph picture description in forming the appearance of a glyph.

9. A method according to claim 7, wherein said compositing operator applied to a plurality of said components performs a selection of one of the components during rendering, the determinant of the selection being chosen by the user of the font.

10. A method according to claims 7, wherein a compositing operator acting on two components performs an operation on the color and/or transparency values, said operation being selected from the group consisting of OVER, IN, OUT, ATOP, XOR, PLUS, MINUS, NOT-XOR, AND, NAND, OR, NOR, NOT, CLEAR, SET, and COPY.

11. A method according to claim 7, wherein said compositing operator acting on a single said component performs a mapping of the color(s) of said component during

rendering, the determinant of the mapping being a characteristic of the font and having the properties described in claim 5.

12. A method according to claim 7, wherein said component of said glyph picture description contains the shape of an outline.

13. A method according to claim 12, wherein said outline is one of:

- (a) filled;
- (b) stroked using a particular type of stroke style; or
- (c) both filled and stroked.

14. A method according to claim 7, wherein said component of said glyph contains the data required to represent a natural or synthesised image, and said image contains color pixel values in any gamut recognisable by the rendering process.

15. A method according to claim 1, wherein said glyph contains said characteristics that apply to one or more of said components of the glyph, said components including:

- (a) color of the components,
- (b) amount of transparency,
- (c) method used to fill the shapes,
- (d) width of the line stroking the shapes,
- (e) style of the shape of the join when two lines meet,
- (f) style of the cap on the end of lines,
- (g) limit on the length of the meter when a metered join is present,
- (h) style of the dash pattern that may be applied to the outline of the shape.

16. A method according to claim 1, wherein said glyph containing said component refers to the shape of another glyph wherein said glyph is taken from another font.

17. A font structure for use in an image creation system comprising a series of characters wherein each character is made up of a customisable glyph structure, said glyph structure further comprising a series of graphical objects which can be composited together in a predetermined order.

18. A method of creating a series of font characters on a computer system comprising:

- providing a series of font outlines and source artwork;
- providing a series of manipulation tools for the manipulation of aspects of said outlines and artwork;
- providing for the creation of substantially arbitrarily complex font structures from said outlines, artwork and manipulation tools; and

creating said series of font characters through the application of said complex font structures to each of a base font outline in said series of font characters.

19. A method according to claim 18, wherein said complex font structures comprise a graphical expression tree of operations to be performed in the creation of a font.

20. A method according to claim 19, wherein said graphical expression tree includes an outline of a font.

21. A method according to claim 18, wherein said manipulation tools include tools for distorting, replacing or compositing the outline of a font.



**22.** A method according to claim 18, wherein said manipulation tools include tools for application of morphological effects to said font outlines.

**23.** A method according to claim 18, wherein said manipulation tools include tools for application of non-morphological effects to said font outlines.

**24.** Apparatus configured to implement the method of according to claim 18.

**25.** A method of creating a font for a plurality of reproducible characters, said method including the steps of:

- (a) providing a plurality of glyphs which together define outlines of said characters having shape characteristics of said font;
- (b) establishing a plurality of records of font attributes;
- (c) associating (first) selected ones of said records with (second) selected ones of said glyphs; and
- (d) manipulating said first selected records to alter a reproduction of said second selected glyphs and hence characters reproduced therefrom.

**26.** A method according to claim 25, wherein said font attributes include characteristics of reproduction of said outlines and filling material for reproduction in association within said outlines.

**27.** A method according to claim 26, wherein step (a) includes providing a plurality of default glyphs associated with a font having first shape characteristics and step (d) includes manipulating said default glyphs to form a further font having second shape characteristics.

**28.** A method according to claim 27, wherein said manipulating said glyphs includes altering at least one of color, an opacity a stroke width, a continuity of a stroke, a shape of a glyph, and a joining between any two glyphs.

**29.** A method according to claim 26, wherein step (b) includes providing a plurality of default attributes associated with said filling material of a font having first fill characteristics and step (d) includes manipulating said default attributes of said filling material to form a further font having second fill characteristics.

**30.** A method according to claim 29, wherein said manipulating said filling material includes altering at least one of a color and an opacity of said filling material.

**31.** A method according to claim 30, wherein said altering includes forming a blend between glyphs.

**32.** A method according to claim 31, wherein said altering includes compositing colors together to form an image within said filling material.

**33.** A method according to claim 26, wherein outlines and filling material for each said character in said font are defined by at least one graphical expression tree including at least one operator acting upon at least one said attribute and said character is reproducible by rendering said expression tree.

**34.** A method according to claim 33, wherein said one attribute includes a predefined image, said predefined image being at least one of a pixel-data image or a graphic object image.

**35.** A method according to claim 25, wherein said attributes are selected from the group consisting of color, opacity, draw style, stroke width, line joining style, line cap style, meter limit, dash pattern and choice number.

**36.** A method according to claim 35, wherein said choice number is used to augment said font with image data.

**37.** A data structure for a font having a plurality of reproducible characters, said data structure comprising:

a plurality of glyphs each of which contribute to at least a shape of one of said characters;

each said glyph having a plurality of attributes which contribute to a reproduction of said glyph in corresponding ones of said characters, said attributes being alterable to thereby modify a reproduction of said glyph in said corresponding characters.

**38.** A method of manipulating a font having a plurality of reproducible characters in a computer system, said font being described by a plurality of glyphs each of which contribute to at least a shape of one of said characters;

each said glyph having a plurality of attributes which contribute to a reproduction of said glyph in corresponding ones of said characters, said attributes being alterable to thereby modify a reproduction of said glyph in said corresponding characters, said method comprising the steps of:

- (a) retrieving data corresponding to a predetermined default font and characters associated therewith including corresponding said glyphs and attributes;
- (b) manipulating selected ones of said attributes associated with selected ones of said glyphs to alter said default font to provide a second font; and
- (c) reproducing at least one character of said second font.

**39.** A method according to claim 38, wherein step (a) includes reproducing in a first manner at least one character of said default font, and step (b) includes interactively altering the default font and reproducing said altered font in said first manner until a final font is formed, and step (c) includes reproducing said final font in a second manner.

**40.** A method according to claim 39, wherein reproducing in said first manner includes displaying said character on a video display associated with said computer system.

**41.** A method according to claim 39, wherein reproducing in said second manner includes at least one of displaying said character on a video display associated with said computer system, printing said character with a printer associated with said system, or recording said character in a form suitable for subsequent reproduction.

**42.** A computer readable medium incorporating a computer program product having a series of instructions interpretable by a computer for creating a font for a plurality of characters, said medium including

means for providing a plurality of glyphs which together define outlines of said characters having shape characteristics of said font;

means for establishing a plurality of records of font attributes;

means for associating (first) selected ones of said records with (second) selected ones of said glyphs; and

means for manipulating said first selected records to alter a reproduction of said second selected glyphs and hence characters reproduced therefrom.

**43.** Apparatus for creating a font for a plurality of characters, said apparatus comprising:

means for providing a plurality of glyphs which together define outlines of said characters having shape characteristics of said font;

means for establishing a plurality of records of font attributes;

means for associating (first) selected ones of said records with (second) selected ones of said glyphs; and

means for manipulating said first selected records to alter a reproduction of said second selected glyphs and hence characters reproduced therefrom.

\* \* \* \* \*

# Visualizing Application Behavior on Superscalar Processors

Chris Stolte, Robert Bosch, Pat Hanrahan, and Mendel Rosenblum  
Computer Science Department  
Stanford University

## Abstract

The advent of superscalar processors with out-of-order execution makes it increasingly difficult to determine how well an application is utilizing the processor and how to adapt the application to improve its performance. In this paper, we describe a visualization system for the analysis of application behavior on superscalar processors. Our system provides an overview-plus-detail display of the application's execution. A timeline view of pipeline performance data shows the overall utilization of the pipeline, indicating regions of poor instruction throughput. This information is displayed using multiple time scales, enabling the user to drill down from a high-level application overview to a focus region of hundreds of cycles. This region of interest is displayed in detail using an animated cycle-by-cycle view of the execution. This view shows how instructions are reordered and executed and how functional units are being utilized. Additional context views correlate instructions in this detailed view with the relevant source code for the application. This allows the user to discover the root cause of the poor pipeline utilization and make changes to the application to improve its performance.

This visualization system can be easily configured to display a variety of processor models and configurations. We demonstrate it for both the MXS and MMIX processor models.

**Keywords:** Computer systems visualization, visualization systems, superscalar processors.

## 1 INTRODUCTION

The processing power of microprocessors has undergone unprecedented growth in the last decade [5]. Desktop computers produced today outperform supercomputers developed ten years ago. To achieve these performance enhancements, mainstream microprocessors such as the Intel Pentium Pro [11] and the MIPS R10000 [15] employ complex pipelines with out-of-order execution, speculation and rename registers.

The implementation techniques used by these processors are intended to be invisible to the programmer. This is true from the standpoint of correctness: application writers need not know the details of the processor implementation to write code that executes correctly. In order to write code that runs *well*, however, programmers need an understanding of their applications' interactions with the processor pipeline. While optimizing compilers

aid in producing compiled code that can take advantage of these powerful processors, they are unable to leverage semantic knowledge about the application in performing their optimizations. Changes made to the code structure of an application by the programmer can increase the instruction-level parallelism that a processor can exploit, resulting in increased performance.

However, because of the complexity of these processors, few software developers understand the interactions between their applications and the processor pipeline. The analysis of application behavior on superscalar processors is complicated by several factors:

- **Having to look at the details.** Many different events can cause poor utilization of the processor pipeline: contention for functional units, data dependencies between instructions, and branching are examples. High-level statistics can indicate that these hazards exist within an application, but they cannot indicate when, where, or why these events are occurring. Understanding specific hazards requires a detailed examination of pipeline behavior at the granularity of individual instructions.
- **Having to know where to look.** Modern processors can execute hundreds of millions of instructions in a single second. Therefore, it is not feasible to browse through either a trace file or detailed visualization of an application's entire execution searching for areas of poor performance. High-level performance overviews of the execution are required to identify regions of interest before detailed visualizations can be used for analysis.
- **Having to have context.** Most programmers think in terms of source code, not in terms of individual instructions. In order to modify their applications to enhance performance, programmers need to be able to correlate instructions in the pipeline with the application's source code.

We have developed a visualization system that addresses all of these issues. Our system consists of three displays: a timeline view of pipeline performance statistics, an animated cycle-by-cycle view of instructions in the pipeline, and a source code view that maps instructions back to lines of code. These views combine to provide an overview-plus-detail [13] representation of the pipeline, enabling the effective analysis of applications. A programmer can utilize the timeline view to observe the time-varying behavior of the pipeline and identify regions of execution where events of interest (such as poor pipeline utilization) occur. The detailed pipeline view can then be used to display and animate the flow of instructions through the pipeline, providing an understanding of why the pipeline is stalling. Finally, the source code view can be used to correlate the problematic instruction sequences with source code, where changes may be made to improve application performance. In addition to program analysis, this visualization system is also useful for several other tasks, including compiler analysis, hardware design, and processor simulator development.

The flexibility of our system enables us to visualize several different processor models, as well as a variety of configurations of a particular processor model. In this paper, we include visualizations of the MXS [1] simulator and two configurations of the MMIX [8] processor model. While our current focus is on the study of processor pipelines, this system could be extended to display other types of pipelines, such as manufacturing assembly lines and graphics pipelines.

## 2 RELATED WORK

Although there are many systems available for high-level analysis of application performance, there are few systems available for this detailed visualization of application execution on superscalar processors. Existing systems include DLXview [3], VMW [2], BRAT [12], and the Intel Pentium Pro tutorial [11].

DLXview [3], an interactive pipeline simulator for the DLX instruction set architecture [5], provides a visual, interactive environment that explains the detailed workings of a pipelined processor. Performance evaluation is a secondary goal of their system: their focus is on the pedagogical nature of visualization. For performance analysis purposes, the pipeline displays of DLXview provide too much detail without enough overall context.

The Visualization-based Microarchitecture Workbench (VMW) [2] is a more complete system for the visualization of superscalar processors. This system was developed with the dual goals of aiding processor designers and providing support to software developers trying to quantify application performance. However, there are several disadvantages to the visualizations and animation techniques used by the system. VMW provides very limited high-level information on application performance, and it is difficult to correlate this information with the detailed views. Animation is used to depict cycle-by-cycle execution, but the animation is not continuous – it consists of sequential snapshots of processor state. While we initially used this approach, we found this animation technique was both difficult to follow and detrimental to understanding the instruction flow.

During the development of the PowerPC, IBM used a simulation tool called the Basic RISC Architecture Timer (BRAT) [12] to study design trade-offs. BRAT provides a graphical interface that allows the user to step through trace files, displaying the processor state at each cycle. BRAT provides only the single, detailed view of the processor state and does not utilize animation in the visualization. Like VMW, this visualization is tightly integrated with the simulator and thus not a general-purpose tool.

Intel distributes an animated tutorial [11] that illustrates the techniques the Pentium Pro processor utilizes to improve performance. Similar to DLXview, the pedagogical intent of this tutorial has resulted in a different design than our system. The tutorial provides a limited cycle-by-cycle view of the instructions in the pipeline with explanatory annotations. No contextual performance data or source code displays are provided.

## 3 BACKGROUND

To provide a context for our visualization, we begin by describing the salient characteristics of superscalar processors that impact application performance. We first introduce the major techniques that superscalar processors use to improve performance, and then explain the types of events that can cause a processor pipeline to be underutilized.

Given a fixed instruction set architecture, a reasonable measure of a processor's performance is the throughput – that is, the number of instructions that complete execution and exit the pipeline in a given period of time. Modern microprocessors utilize several techniques to improve their throughput:

- **Pipelining.** Pipelining overlaps the execution of multiple instructions within a functional unit, much like an assembly line overlaps the steps in the construction of a product. For example, a single-stage floating point unit might require 60 cycles to complete execution of a single divide instruction. If this functional unit were pipelined into six stages of 10 cycles apiece, the unit would be able to process multiple divide instructions at once (with each stage working on a particular piece of the computation). While it would still require 60 cycles to compute a single divide, the pipelined functional unit would produce a result every 10 cycles when performing a series of divide instructions.
- **Multiple Functional Units.** Superscalar processors include multiple functional units, such as arithmetic logic units and floating-point units. This enables the processor to exploit instruction-level parallelism (ILP), executing several independent instructions concurrently. However, some instructions cannot be executed in parallel because one of the instructions produces a result that is used by the other. These instructions are termed *dependent*.
- **Out-of-Order Execution.** In order to improve functional unit utilization, many superscalar processors execute instructions out of order. This allows a larger set of instructions to be considered for execution, and thus exposes more ILP. Out-of-order execution can improve throughput if the next instruction to be sequentially executed cannot utilize any of the currently available functional units or is dependent on another instruction. Although instructions may be executed out of order, they must *graduate* (exit the pipeline) in their original program order to preserve sequential execution semantics. The reordering of instructions is accomplished in the *reorder buffer*, where completed instructions must wait for all preceding instructions to graduate before they may exit the pipeline.
- **Speculation.** Rather than halting execution when a branch instruction is encountered until the branch condition is computed, most processors will continue to fetch and execute instructions by predicting the result of the branch. If the processor speculates correctly, throughput is maintained and execution continues normally. Otherwise, the speculated instructions are *squashed* and their results are discarded.

Despite the use of these techniques, superscalar processors are often unable to achieve maximum throughput. There are many possible causes for underutilization of the pipeline.

When there are not enough functional units to exploit the ILP available in a code sequence, instructions must wait for a unit to become available before they can execute. These *structural hazards* often occur in code that is biased towards a particular type of instruction, such as floating-point instructions. The functional unit for those instructions will be consistently full, and the other units will often remain empty for lack of instructions. Consequently, the throughput of the processor is limited to the throughput of the critical functional unit alone.

*Dependencies* prevent instructions from executing in parallel. Out-of-order execution enables the pipeline to continue execution in the face of individual dependencies; however, if a code se-

quence includes enough dependencies, the lack of ILP will limit pipeline throughput.

Speculative execution can impact throughput in two ways. First, most processors cannot speculate through more than four or five branches at once. Once this *deep speculation* is reached, the processor cannot speculate through subsequent branch instructions. This forces the pipeline to stop fetching instructions until one of the pending branches is resolved. Second, processors do not always predict the result of a branch correctly. When *branch misprediction* occurs, throughput suffers since the incorrectly speculated instructions must be squashed from the pipeline.

Because main memory accesses often require hundreds of cycles to complete, *memory stall* can have a major impact on pipeline performance. When an instruction cache miss takes place, the processor cannot fetch instructions into the pipeline until the next sequence of instructions is retrieved from memory. The resulting lack of instructions in the pipeline reduces the processor throughput. Misses to the data cache increase the effective execution time of load and store instructions, since they must wait for the memory access to complete before they can graduate. This delays the execution of any dependent instructions, and eventually stalls the pipeline by preventing subsequent instructions from graduating.

Finally, some instructions, such as traps and memory barrier instructions, require *sequential execution*, forcing the pipeline to be emptied of all other instructions before they can execute. This has an obvious detrimental effect on throughput.

Although high-level visualizations can indicate that these events are occurring during the execution of an application, only detailed visualizations can reveal the instruction flow and dependencies that are responsible for these performance bottlenecks. This detailed knowledge is critical for adapting the application to improve the performance.

## 4 VISUALIZATION ENVIRONMENT

The pipeline visualization system discussed in this paper was developed using Rivet. Rivet is a visualization environment we are developing to support the rapid prototyping of visualizations for the exploration and understanding of real world problems, with an emphasis on the analysis and visualization of computer systems. Several attributes of Rivet were particularly important for the development of this visualization system.

**Flexibility.** Rivet provides flexibility through a compositional architecture. Components such as data objects and visual primitives, written in C++ and OpenGL, are designed to be *composed* to form objects with greater functionality. Primitives and objects also export an interface to a scripting language such as Tcl, which allows them to be composed further to create sophisticated, interactive visualizations.

One of our design goals for the pipeline visualization system was to make it easily adaptable to many processor models. To support this, our processor pipeline display is composed from two major classes of visual primitives, *containers* and *pipes*, both written in C++. We use the Tcl scripting language to combine these primitives into higher-level building blocks: the functional units, stages, and data paths of the processor pipeline. We then combine these objects according to the configuration of the particular processor model under study to represent the entire pipeline.

The decomposition of the pipeline into its constituent elements enables us to easily adapt the layout to represent a variety of processor models with different pipeline organizations. It also enables

us to easily configure the visualization according to the parameters of a particular processor model, such as the number of functional units or the size of the reorder buffer.

**Aggregation.** The complexity of computer systems demands that a visualization environment be able to efficiently manage and display large data sets. To simplify this task, Rivet provides built-in data management objects that support data aggregation. Data is collected at a fine granularity over a long period; it is then built into an aggregation structure that includes both the raw data and smaller, less detailed aggregates. When displaying the data, Rivet chooses the appropriate data resolution based on the time window to be displayed and the available screen space.

The study of superscalar processors requires large volumes of data be collected and visualized. Our experimental runs often generate data for hundreds of thousands of execution cycles. For each cycle, information about any instruction that changes state must be collected. The use of Rivet aggregation structures enables us to explore this data from a high-level overview down to individual data elements.

**Animation.** Animation is a core service provided by Rivet. This support includes the ability to request timed callbacks to visual primitives and path interpolation for a variety of animation paths. The Rivet redraw mechanism supports incremental redraw of visual primitives, important for efficient animation of objects.

Animation is crucial for understanding the cycle-by-cycle behavior of the pipeline. Our original pipeline implementation simply displayed the state of the pipeline at a particular cycle with no visual transitions between cycles. Without the visual cues provided by animation, we found it very difficult to track instructions as they advanced through the pipeline.

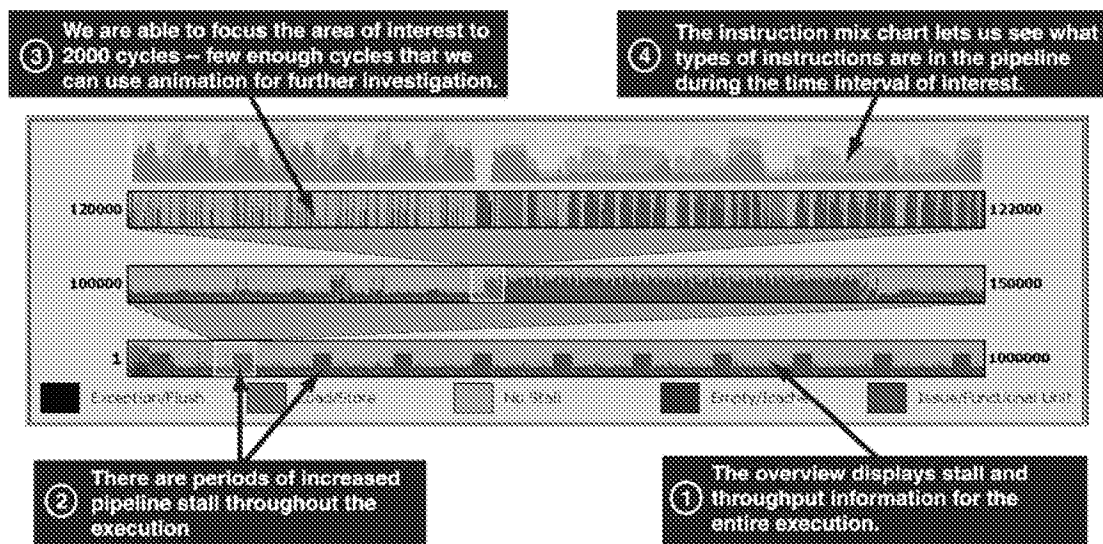
## 5 PIPELINE VISUALIZATION SYSTEM

Our pipeline visualization combines three major components to provide an overview-plus-detail display of application execution. We first describe each of the components of the system, and then present an example showing how the system is used to understand application behavior.

### 5.1 Timeline View: Finding Problems

The first task in understanding the behavior of an application on a processor is to examine an overview of the application's execution to locate regions of interest. The *timeline view*, shown in Figure 1, utilizes a multi-tiered strip chart to display overall pipeline performance information at multiple levels of detail. The bottom tier shows data collected over the entire execution of the application. The user interactively selects regions of interest in each tier, which are expanded and displayed in the next tier. This visualization exploits the aggregation mechanism described in Section 4: the data in each tier is displayed at the highest resolution possible, determined by the number of horizontal pixels available for display.

The multi-tiered strip chart is used to indicate the reasons that the pipeline was unable to achieve full throughput on a particular cycle (or range of cycles in the aggregated displays). In a superscalar processor, throughput is lost whenever the pipeline fails to graduate a full complement of instructions from the pipeline in a given cycle. Because instructions must graduate in order, we attribute this pipeline graduation stall to the instruction at the head of the graduation queue (i.e. the oldest instruction in the pipeline). The reasons for failure to achieve full pipeline throughput can be classified into the following categories:



**Figure 1:** Investigation of an application's processor pipeline behavior typically begins by examining high-level performance characteristics. The timeline view provides a multi-tiered strip chart for the exploration of this data. Pipeline throughput statistics for the entire execution are shown on the bottom tier of the strip chart, with pipeline stall time classified by cause (as shown in the legend at the bottom of the window). The yellow panes are used to select time intervals of interest in each tier, which are displayed in more detail in the next tier. Directly above the multi-tiered chart is a simple strip chart that shows the instruction mix in the pipeline during the time region of interest: load/store (green), floating-point (pink), branch (yellow) and integer (cyan). This strip chart serves to relate this high-level view to the detailed pipeline view.

- **Empty/ICache.** An instruction cache miss is preventing any instructions from being fetched from memory, so the pipeline is completely empty (and there is no head of the instruction queue to blame for the stall).
- **Exception/Flush.** Either an exception occurred or an instruction in the pipeline requires sequential execution. In either case, the pipeline must be flushed before continuing execution, again leaving the pipeline empty until instruction fetch resumes.
- **Load/Store.** The head of the graduation queue is a memory load or store operation that is waiting for data to be retrieved from the memory system.
- **Issue/Functional Unit.** The head of the graduation queue is either waiting to be issued into a functional unit due to a structural hazard or is still being processed by a functional unit.

In addition to the pipeline stall information, the timeline view includes a second chart that displays the mix of instructions in the pipeline. This chart classifies instructions by functional unit and shows the instruction mix during the same time window as the top tier of the multi-tiered strip chart. By relating the reasons for pipeline stall to the instructions in the pipeline, this display serves as a 'bridge' between the timeline view and the pipeline view.

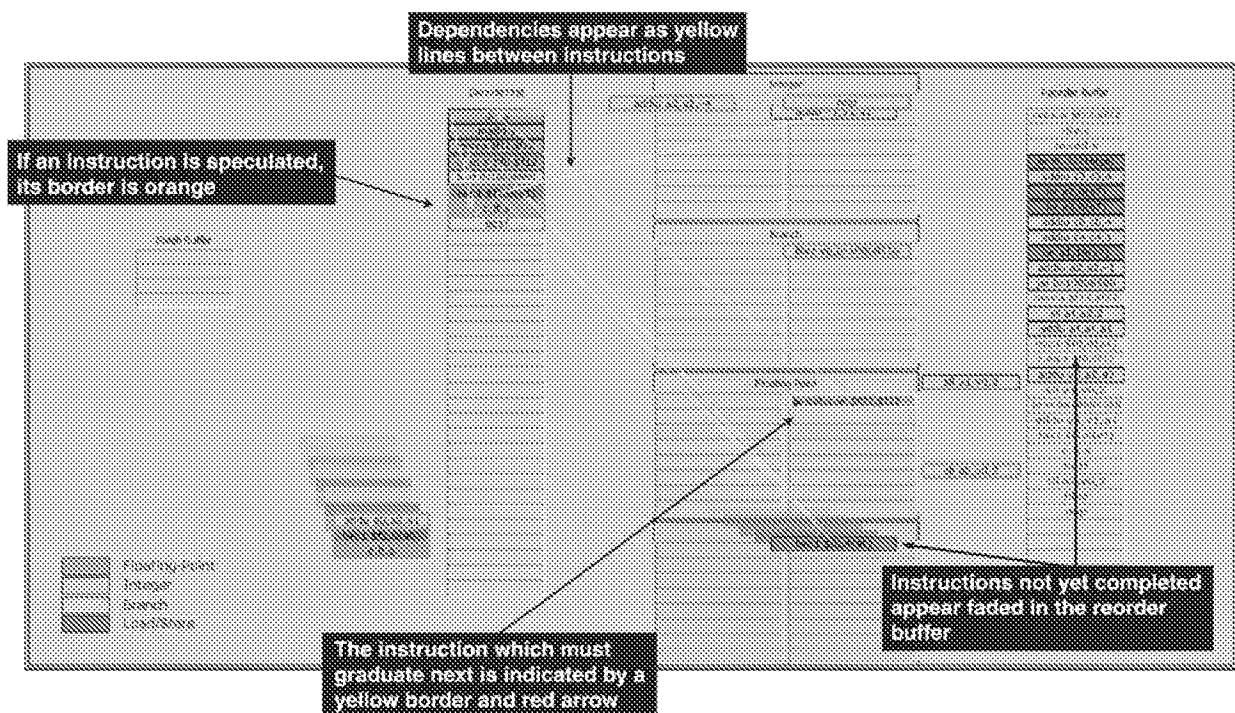
## 5.2 Pipeline View: Identifying Problems

The pipeline view is illustrated in Figure 2. This visualization shows the state of all instructions in the pipeline at a particular cycle and animates instructions as they progress through the pipeline. The animation techniques used in this view are similar to those used for program debugging in [10] and algorithm animation in [14].

Pipeline stages and functional units appear as large rectangular regions with numerous instruction slots. Each stage is represented as a single container, with the number of slots indicating the capacity of the container. Functional units are composed of one or more containers, since these units may themselves be composed of multiple pipeline stages. The functional units are color-coded using the same color scheme as the instruction mix strip chart. The layout of the pipeline is interactively configurable. At any time, the user can reorder the layout of the functional units or resize the stages and functional units to focus on a portion of the processor pipeline.

Instructions in the pipeline are depicted as rectangular glyphs. The glyphs encode several pieces of information about the instructions in their visual representation. The fill color of the rectangle matches the color of functional unit responsible for execution of the instruction. The glyph contains text identifying the instruction; depending on the space available, either the opcode mnemonic or the full instruction disassembly (including both the mnemonic and the arguments) is displayed. The border color of the instruction conveys additional information. If the instruction has been issued speculatively and the branch condition is still unresolved, the border of the instruction is orange. If the instruction was issued as a result of incorrect speculation and will subsequently be squashed, it is drawn with a red border. The head of the graduation queue always has a yellow border, and a red triangle appears next to the text of this instruction.

Dependencies between instructions in the pipeline are displayed as yellow lines appearing between the two instructions. Since a large number of dependencies may be present in the pipeline, the user can selectively filter or disable this feature. To filter this display, the user selects with the mouse the instruction for which dependencies should be drawn.



**Figure 2:** The pipeline view shows all instructions in the pipeline at a particular point in time. Pipeline stages and functional units appear as large rectangular regions with numerous instruction slots. This processor has a four-stage pipeline – fetch, decode, execute and reorder – arranged from left to right in the figure. Instructions are portrayed as rectangular glyphs, color-coded to indicate their functional unit and labeled to identify their opcode. Additional information about the state of the instruction is encoded in the border color of the glyph. User-controlled animation is used to show the behavior of instructions as they advance through the pipeline. This figure illustrates the animated transition between two cycles of execution of a graphics application executing on the MXS processor model. The pipeline can fetch and graduate up to four instructions per cycle. However, in this case the processor was unable to graduate any instructions because the head of the graduation queue is still being executed in the floating-point functional unit.

With the exception of the reorder buffer, the pipeline stages order instructions by age: instructions enter at the bottom of the stage and move upward to replace instructions that have exited the stage. In the reorder buffer, instructions are shown in graduation order, with the head of the graduation queue at the top of the buffer. The reorder buffer leaves empty slots for instructions that are executing in the pipeline but have not yet completed. These slots contain a grayed-out text label of the instruction, enabling the slots to be correlated with the instructions in the pipeline.

The user controls the pipeline animation using controls similar to those used to control a VCR. The controls enable the user to single-step, animate, or jump through the animation. The animation may be run either forward or backward, and the speed is variable and under user control. The user can also use a vernier on the instruction mix strip chart to jump directly to a particular cycle.

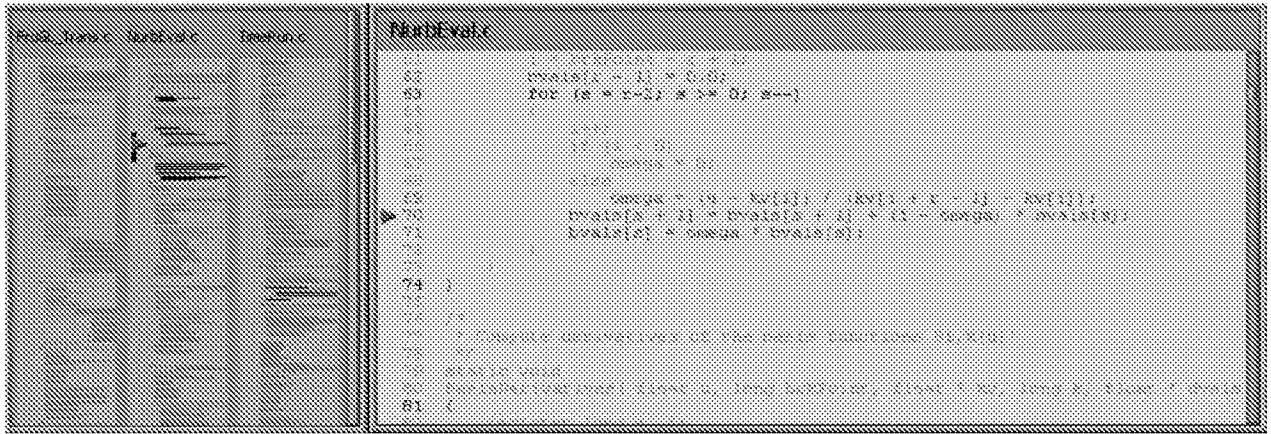
This visualization can be used to understand the precise nature of the observed pipeline hazards. The user can animate through cycles of interest and visually identify for each cycle the hazards that are occurring. Figure C-1 (color plate) illustrates the visual characteristics of several of the major types of hazards. By interacting with the pipeline view, the user can observe the instruction sequences that are responsible for underutilizing the pipeline and understand the reasons for their poor performance. In order for this information to be useful, however, it must be related back to the source code of the application.

### 5.3 Source Code View: Providing Context

Once the major regions of poor performance and their causes have been discovered, the user must determine if the application can be altered to improve pipeline utilization. The final component of our system, the *source code view*, allows the user to correlate the high-level performance data and detailed animation views with the application's source code. This view is shown in Figure 3.

This visualization, modeled on the *SeeSoft* system [4], displays “bird’s-eye” overview representations of the source files in the application. Each line of source code is represented by a single-pixel horizontal bar; the length and indentation of each bar is proportional to the actual indentation and length of the line in the source. The user can select a region of a source file in the overview to be displayed as full source code text in a separate window, as shown on the right side of the figure.

Both windows in the source code view highlight relevant lines of code. Lines that are executed at some point during the time window of interest in the timeline view are drawn in black, and lines that are being executed in the pipeline view are highlighted in red. As in the pipeline view, a red arrow is displayed next to the line of source code that contains the instruction at the head of the graduation queue.



**Figure 3:** The source code view, which relates the instructions in the timeline’s region of interest to the source code corresponding to these instructions. The left panel provides a bird’s eye view of the source, and the right panel shows a portion of one of the source files (indicated by the vertical bar in the left panel). Both views are color-coded to highlight instructions in the region of interest. Black indicates that the line of code is executed somewhere in the timeline region and red indicates that the line is being executed in the detailed pipeline view. A red arrow indicates the instruction at the head of the graduation queue.

## 5.4 Visualizing the MXS Pipeline

We now provide an example of how the three components of the system can be used together to understand the behavior of an application. For data collection, we use the MXS [1] superscalar processor model. This model implements the instruction set architecture used in the MIPS R10000 [15] processor. MXS has been incorporated into the SimOS complete machine simulator [6], enabling us to study the pipeline utilization of realistic workloads. In this example, we describe the analysis and visualization of a graphics application executing for one million cycles. Figure 4 and Figure C-2 (color plate) show a snapshot of the visualization of this data.

We begin the analysis by looking at the timeline view of the execution. The bottom tier of the multi-tiered strip chart shows a periodic execution pattern. Of interest are the phases where we see a significant increase in processor stall time. The chart shows that throughput is limited in these phases because the head of the graduation queue is still executing in a functional unit. There are several reasons why this might occur, such as an unbalanced mix of instructions or a large number of dependencies.

We use the multi-tiered strip chart to zoom in on the transition from high throughput to low throughput. As we zoom to a view of 50,000 cycles, the high-level pattern becomes less apparent but we can still see the two distinct phases of execution. We zoom further to a window of 2000 cycles centered on the phase transition. By comparing the throughput chart with the instruction mix chart, we discover that the bulk of the processor stall time corresponds to periods of heavy floating-point activity in the pipeline.

We investigate this further using the pipeline view. We animate this region of the execution and observe the instructions as they travel through the pipeline. The pipeline view shows a representative stage of the animation. By observing the animation, we are quickly able to see why the pipeline is suffering from poor throughput: there is a cascading dependency chain between nearly all of the instructions in the decode unit. This complete lack of instruction-level parallelism forces the pipeline to process instructions in a sequential fashion. Even worse, the instruction window is dominated by floating-point instructions, including

operations with long execution latencies like the divide (the instruction in the floating-point unit in the figure). As a result, there are few (if any) instructions available to graduate per cycle.

We can use the source code view to correlate this pipeline behavior with the application’s source code. We see in the source code that the application is executing a tight loop of floating point arithmetic. With this information, the programmer can now attempt to restructure the code to reduce the number of dependencies or interleave other code into the loop to better utilize the processor.

## 6 DISCUSSION

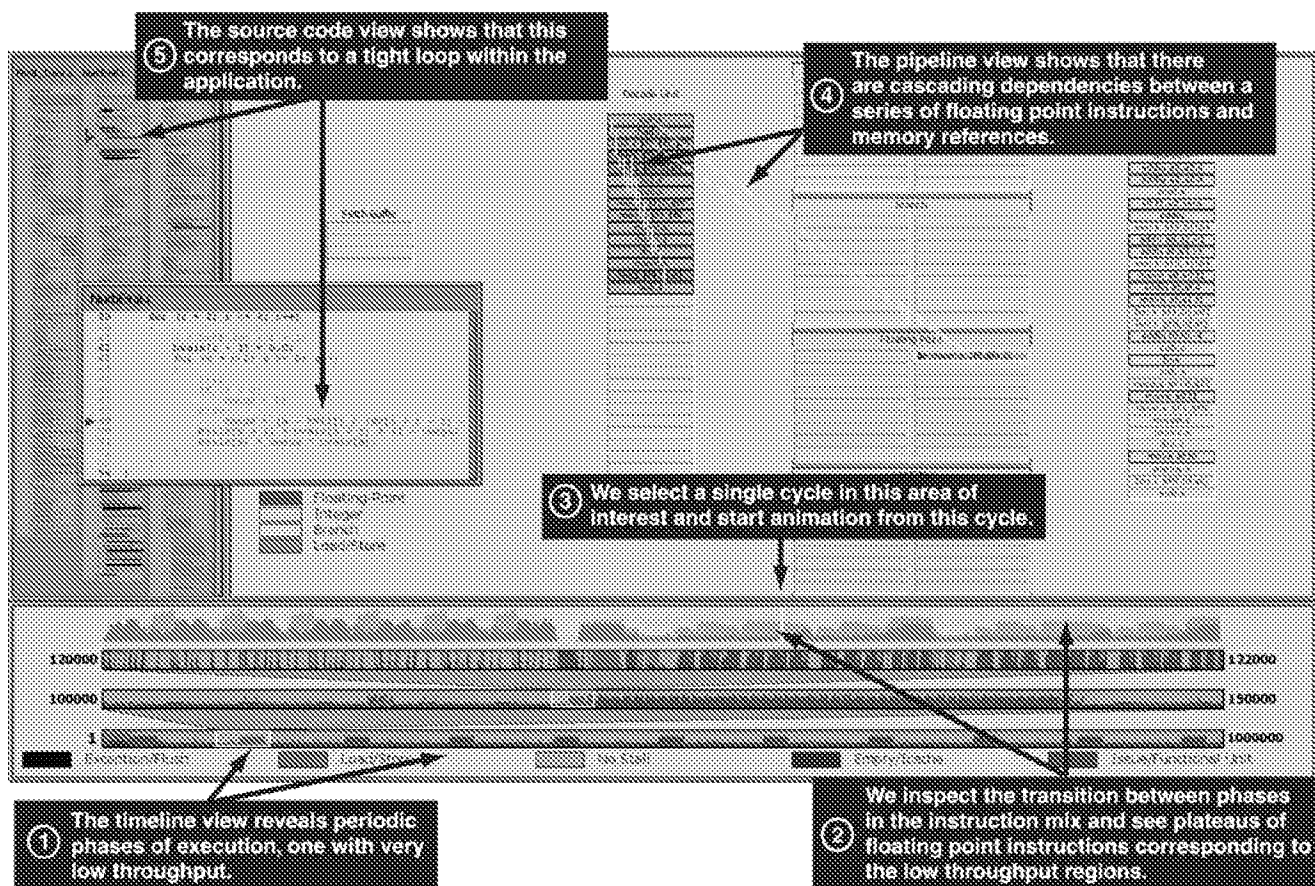
We have presented the pipeline visualization system in the context of one specific use – the understanding of application behavior on a superscalar processor. However, there are several other important uses for this visualization system.

**Compiler Design.** One of the major research areas in compiler design is code optimization. Research is being done to study the effectiveness of the code optimization techniques that have been developed and to discover and implement additional optimizations. In particular, with the popularity of superscalar processors, compiler writers are striving to maximize the amount of instruction-level parallelism in compiled code in order to make full use of processor resources. By exposing the detailed behavior of the processor pipeline, our visualization system can be used to study the effectiveness of compiler optimizations and suggest code sequences that would benefit from further optimization.

**Hardware Design.** When designing new processors, hardware architects need to understand the demands that applications used by their target markets will place on the processor. By using visualization to study the behavior of important commercial applications on existing processors, they can identify where architectural changes such as additional functional units or pipelining would be beneficial.

As a simple example of this use of our system, Figure 5 shows two visualizations of a prime number generator running on MMIX [8], an architecture being developed by Donald Knuth for his series of books, *The Art of Computer Programming* [9]. The





**Figure 4: The complete processor pipeline visualization system displaying one million cycles of execution. The timeline view shows a periodic behavior, with alternating sections of high and low processor stall. The chart is zoomed in on the region of low utilization. The pipeline view shows that the instruction sequences in this window are highly dependent on one another, with very little instruction-level parallelism available to be exploited. The source code view shows the code segment corresponding to this phase of execution – a tight floating-point loop with dependencies both within the loop and across iterations.**

example shows how pipelining the divide functional unit can improve the performance of this application.

**Simulator Development.** Simulation is a powerful technique for the understanding of computer systems such as microprocessors. During the design of new processors, simulators are developed to explore the processor design space and validate architectural decisions. Simulators are also used for performance analysis of existing applications and processors. However, because of their complexity, the development of processor simulators is a challenging and error-prone task.

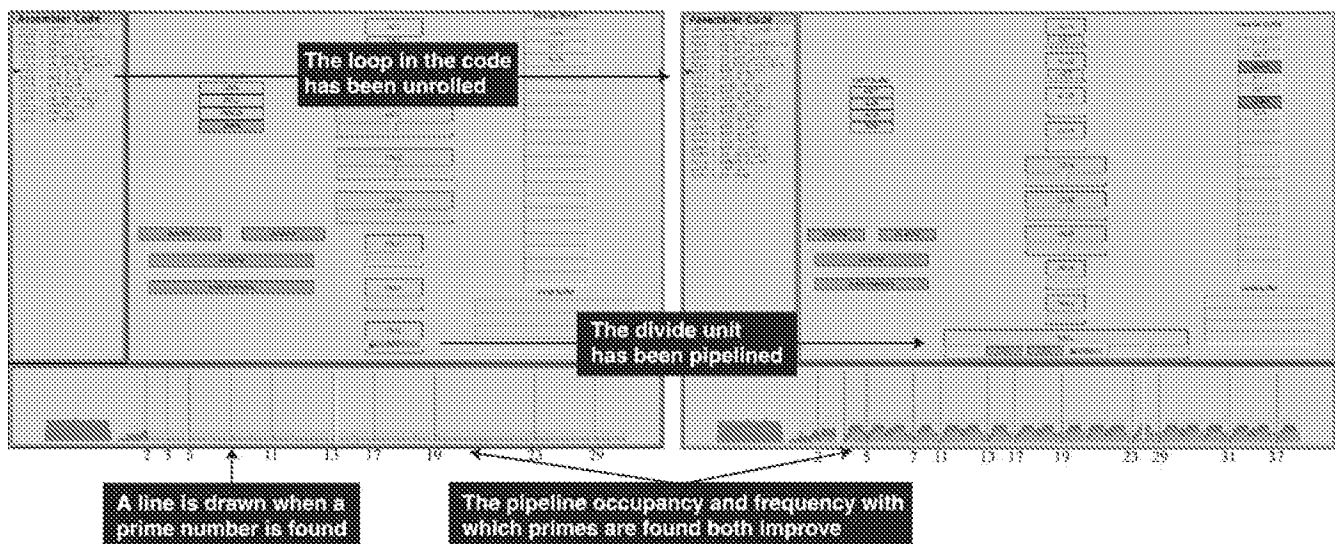
While developing the pipeline visualization system, we uncovered several errors in the MXS and MMIX processor models, many of them timing related. For example, in the original implementation of MXS, it was possible for instructions to advance through all the stages of the processor pipeline in a single cycle. Timing bugs such as these did not affect correctness – simulated programs would still execute correctly – but resulted in timing behavior that was not faithful to the processor model. While the aggregate pipeline statistics obscured these problems, which had existed for some time in the simulators, examination of the timeline view and observation of the pipeline animation made them prominent and enabled us to correct them.

## 7 CONCLUSION

We have presented a system for the visualization of a particularly complex system – superscalar processors. The main goal of our system is the analysis and optimization of application performance on this class of processors. By providing overview-plus-detail displays, the visualization system allows the user to see both high-level performance characteristics and the intricate details of out-of-order execution, speculation and pipelining. We have also described three other uses for our system: hardware design, compiler design and simulator development.

Our future work will build on this system in several ways. First, there are several other attributes of the processor state that we will incorporate into our visualizations, such as register files and write buffers, in order to provide a more complete picture of the pipeline's behavior.

Second, while superscalar architectures currently dominate the market, processors with alternate designs are also being developed in an attempt to maximize performance. For example, the Intel IA-64 architecture [7] uses a “very long instruction word” (VLIW) style of architecture. We intend to extend our visualization system to model these alternate architectures, enabling the exploration of trade-offs between different processor styles.



**Figure 5:** The pipeline visualization system can also be used to study the impact of changes to processor implementations. This figure compares the first 2000 cycles of execution of an MMIX program that calculates the first 500 prime numbers. On the left, an initial implementation of the program is executing on a pipeline with a simple 60-cycle functional unit for divide instructions. On the right, a modified version is executing on a configuration with the divide unit pipelined into six 10-cycle stages. In the modified version of the program, the main loop has been manually unrolled three times to better utilize the pipelined divide unit. To aid the comparison, the system draws a thin gray line in the instruction mix chart when the program finds a prime number. Examining the instruction mix chart for each processor, we can see that the second implementation consistently has more instructions in the pipeline and is progressing more rapidly. The increased amount of pink (floating-point instructions) in the instruction mix chart reflects the fact that the pipelined divide unit is enabling the processor to work on several divide instructions at once.

Finally, although our pipeline visualization system has focused on the study of microprocessors, we would like to explore the application of this system to generalized pipeline systems such as assembly lines and organizational work flow. We expect the overall approach of overview-plus-detail will apply equally well in these areas, and our compositional architecture will enable us to adapt our visualizations to apply to these problem domains.

## Acknowledgments

The authors thank John Gerth for his significant contributions to the design and development of the Rivet system, and Donald Knuth for working with us to develop the visualization of the MMIX processor. We also thank Tamara Munzner, Diane Tang and David Ofelt for both their work reviewing this manuscript and for useful discussions.

## References

- [1] James Bennett and Mike Flynn. "Performance Factors for Superscalar Processors." Technical Report CSL-TR-95-661, Computer Systems Laboratory, Stanford University, February 1995.
- [2] Trung A. Diep and John Paul Shen. "VMW: A Visualization-Based Microarchitecture Workbench." *IEEE Computer* 28(12), December 1995.
- [3] *DLXView*. [online] Available: <<http://yara.ecn.purdue.edu/~teamaaa/dlxview/>>, cited March 1999.
- [4] Stephen G. Eick, Joseph L. Steffen and Eric E. Sumner, Jr. "See-Soft – A Tool for Visualizing Line-Oriented Software Statistics." *IEEE Transactions on Software Engineering*, 18(11):957-968, November 1992.
- [5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kaufmann Publishers, 1996.
- [6] Stephen A. Herrod. "Using Complete Machine Simulation to Understand Computer System Behavior." Ph.D. Thesis, Stanford University, February 1998.
- [7] Intel Corporation. *Merced Processor and IA-64 Architecture*. [online] Available: <<http://developer.intel.com/design/IA64/>>, cited July 1999.
- [8] Donald Knuth. *MMIX 2009: A RISC Computer for the Third Millennium*. [online] Available: <<http://sunburn.stanford.edu/~knuth/mmix.html>>, cited March 1999.
- [9] Donald Knuth. *The Art of Computer Programming*. 3 vols. Reading, MA: Addison-Wesley, 1997-1998.
- [10] Sougata Mukherjee and John T. Stasko. "Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding." *Proceedings of the 15<sup>th</sup> International Conference on Software Engineering*, pp. 456-465, May 1993.
- [11] *Pentium® Pro Processor Microarchitecture Overview Tutorial*. [online] Available: <<http://developer.intel.com/vtune/cbts/pproarch/>>, cited March 1999.
- [12] Ali Poursepanj. "The PowerPC Performance Modeling Methodology." *Communications of the ACM*, 37(6):47-55, June 1994.
- [13] Ben Shneiderman. "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualization." *Proceedings of IEEE Workshop on Visual Languages*, pp. 336-343, 1996.
- [14] John T. Stasko. "Animating Algorithms with XTANGO." *SIGACT News*, 23(2):67-71, Spring 1992.
- [15] Kenneth Yeager. "The MIPS R10000 Superscalar Microprocessor." *IEEE Micro*, 16(2):28-40, April 1996.

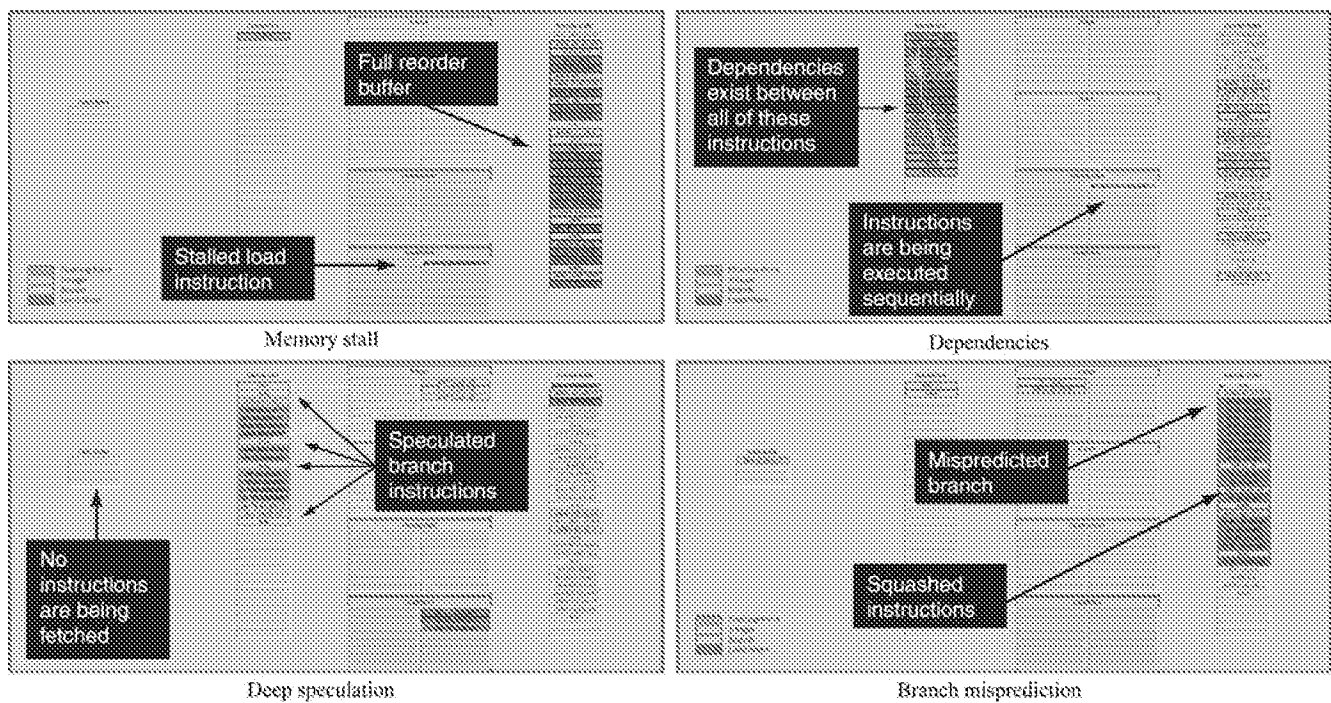


Figure C-1: Snapshots of a program's execution in the pipeline view, demonstrating a variety of reasons for poor pipeline utilization. After using the visualization system for a short time, users can quickly identify these hazards as they occur in the animation.

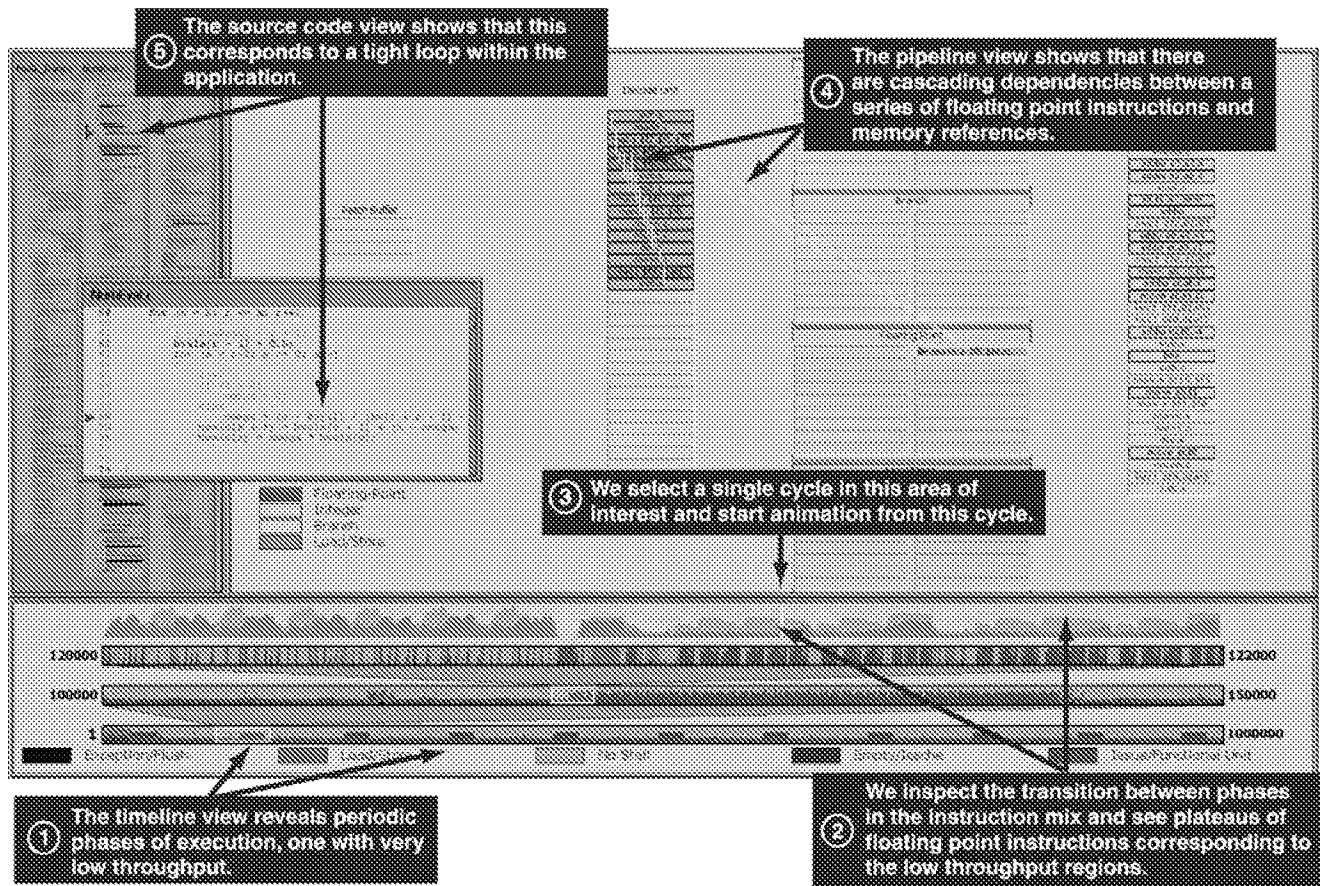


Figure C-2: The complete processor pipeline visualization system displaying one million cycles of execution.